FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Entwicklung einer Online-Visualisierung zu dem Simulationsprogramm NBODY6++ mit Hilfe der Kopplungsbibliothek VISIT

Sonja Dominiczak

FZJ-ZAM-IB-2005-02

Januar 2005

(letzte Änderung: 26.01.2005)

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

Numerische Computersimulationen benötigen ein hohes Maß an Rechenzeit und Speicherplatz. Die direkte Visualisierung von Zwischenergebnissen (Online-Visualisierung) und die direkte Steuerung der Simulation (Computational Steering) können einen Beitrag zur effizienten Nutzung dieser Ressourcen leisten. Mit Hilfe dieser Werkzeuge kann der Benutzer von seinem Arbeitsplatz aus auf falsche oder unerwünschte Ergebnisse reagieren, noch bevor die Simulation vollständig beendet ist. Er kann das Simulationsprogramm abbrechen oder interaktiv Parameter verändern und dadurch die Simulation in eine andere Richtung lenken. Auf diese Weise kann wertvolle Rechenzeit eingespart werden.
In dieser Arbeit wird die Entwicklung der Online-Visualisierung *xnbody* zu dem Simulationsprogramm NBODY6++, speziell für die Anwendung „Simulation von Sternhaufen“ vorgestellt. NBODY6++ ist ein Programm zur Simulation von N-Körper-Problemen, welches für den Einsatz auf Supercomputern parallelisiert wurde. Es wird zur Simulation des dynamischen Verhaltens von Sternhaufen im Rahmen eines wissenschaftlichen Projektes auf dem IBM-Supercomputer des Forschungszentrums Jülich eingesetzt. Es werden die bei der Implementierung verwendeten Konzepte vorgestellt. Insbesondere war es für die Effizienz der Online-Visualisierung erforderlich eine Zwischenspeicherung für die Simulationsdaten zu integrieren. Dadurch können auch zurückliegende Zeitpunkte und Flugbahnen der einzelnen Sterne beobachtet werden.

Numerical computer simulations require a huge amount of computing time and storage. Through interactive monitoring (online-visualization) and direct control of the simulation (computational steering), these resources may be used more efficiently. By applying these tools, the user is able to react to incorrect or unwanted results even before the simulation terminates. In this way, the simulation can be aborted or controlled by interactively modifying parameters and thus save computing time.
This report describes the development of *xnbody*, a new online-visualization-program designed for the simulation of star clusters with NBODY6++. This program has been developed for the simulation of nbody-problems and has been parallelized for the use on supercomputers. The simulator program is being used on the IBM-supercomputer of the Forschungszentrum Jülich for the simulation of the dynamic behavior of star clusters. In this report, the concepts which have been applied in implementing the online-visualization are described. Particularly with regard to the efficiency of the online-visualization, it was necessary to integrate a buffering-mechanism for the simulation data. In this way it is possible to observe earlier data and trajectories of single stars as well.

Kapitel 1

Einleitung

Durch die stetig wachsende Leistungsfähigkeit von Supercomputern gewinnen numerische Computersimulationen auf dem Gebiet des wissenschaftlichen Rechnens immer mehr an Bedeutung. In vielen Bereichen der Wissenschaft lassen sich mit ihrer Hilfe Untersuchungen durchführen, die aus verschiedenen Gründen nicht am realen System durchgeführt werden können. So ist zum Beispiel die Untersuchung und Beobachtung von astrophysikalischen Prozessen am realen System schwierig, da sich die Prozesse über einen sehr langen Zeitraum erstrecken.

Computersimulationen in den Naturwissenschaften nehmen oft ein hohes Maß an Rechenzeit und Speicherplatz in Anspruch. Je höher diese Anforderungen sind, desto wichtiger wird eine effiziente Nutzung dieser Ressourcen. In diesem Zusammenhang sind die Visualisierung von Zwischenergebnissen bei Simulationsrechnungen, *Online-Visualisierung*, und die interaktive Steuerung von Simulationsprogrammen, *Computational Steering*, wichtige Software-Werkzeuge.
Mit Hilfe dieser Werkzeuge kann der Benutzer auf falsche oder unerwünschte Ergebnisse reagieren, noch bevor die Simulation vollständig beendet ist. Er kann das Simulationsprogramm abbrechen oder interaktiv Parameter verändern und dadurch die Simulation in eine andere Richtung lenken. Auf diese Weise kann wertvolle Rechenzeit eingespart werden.

Ziel dieser Arbeit ist die Entwicklung einer Online-Visualisierung zu dem Simulationsprogramm NBODY6++, speziell für die Anwendung „Simulation von Sternhaufen“. NBODY6++ ist ein Programm zur Simulation von N-Körper-Problemen, welches für den Einsatz auf Supercomputern parallelisiert wurde und zur Simulation von Sternhaufen im Rahmen eines wissenschaftlichen Projektes auf dem IBM-Supercomputer der Forschungszentrum Jülich GmbH eingesetzt wird.

Bei der Realisierung einer Online-Visualisierung müssen verschiedene Aspekte berücksichtigt werden: Simulations- und Steering-Daten müssen zwischen der Simulation und der Online-Visualisierung übertragen werden. Dabei ist es wichtig, dass die Online-Visualisierung in der Lage ist, auf Benutzereingaben zu reagieren, während sie auf die Übertragung von Simulationsdaten wartet. Aber auch die Simulationsrechnung soll durch die Kopplung mit der Online-Visualisierung so wenig wie möglich beeinträchtigt werden. Zwischenspeicher auf der Simulations- und Visualisierungsseite tragen zur Effizienz der Kommunikation bei, da durch sie die Häufigkeit der Datenübertragungen reduziert wird. Die Simulationsdaten werden in einem Zwischenspeicher auf der Simulationsseite gesammelt und dann in Blöcken zur Visualisierung übertragen. Dort werden sie wiederum in einem Zwischenspeicher für die Visualisierung bereitgehalten.

Bei der in dieser Arbeit entwickelten Online-Visualisierung zu dem Simulationsprogramm NBODY6++ werden die Positionen und Geschwindigkeiten der Sterne sowie einige weitere Eigenschaften wie zum Beispiel Kräfte, die auf einen Stern wirken, visualisiert. Der Zwischenspeicher auf der

Visualisierungsseite erlaubt eine Navigation zwischen den im Speicher befindlichen Simulationszeitpunkten sowie die Darstellung der Flugbahnen der Sterne. Die Funktionen des Zwischenspeichers müssen an die Eigenschaften der Simulationsdaten angepasst sein. Zum einen werden bei der Simulation nicht zu jedem Simulationszeitpunkt die Daten aller Sterne neu berechnet. Sich langsamer bewegende Sterne werden weniger häufig berechnet, so dass zum gewünschten Zeitpunkt die Daten dieser Sterne eventuell nicht vorliegen. Es ist Aufgabe des Zwischenspeichers, die Daten dieser Sterne anhand von Daten zu früheren Simulationszeitpunkten zu approximieren. Zum anderen werden Sterne, die sehr nahe zusammenkommen, zu einem weiteren Stern zusammengefasst. Dies muss bei der Ermittlung von Flugbahnen berücksichtigt werden.

Neben der Implementierung des Zwischenspeichers ist bei der Realisierung der Online-Visualisierung die graphische Darstellung der Simulationsdaten wesentlich. Die Sterne sollen als Kugeln bzw. Punkte und deren Geschwindigkeiten als Pfeile dargestellt werden. Andere Attribute wie zum Beispiel die Dichte sollen durch Farben bzw. Grössen der Objekte visualisiert werden.

Zuletzt müssen die Darstellungen der Simulationsdaten in eine graphische Benutzeroberfläche integriert werden. Über die graphische Oberfläche kann der Benutzer die Darstellungen an seine Anforderungen anpassen und Steering-Parameter eintragen, durch die dann die Simulation beeinflußt werden kann.

Im folgenden Kapitel werden zunächst Werkzeuge vorgestellt, die zur Realisierung der Online-Visualisierung eingesetzt werden. In Kaptiel 2 werden Eigenschaften des Simulationsprogramms NBODY6++ vorgestellt, die einige Besonderheiten der Simulationsdaten bedingen. In Kapitel 4 werden die Anforderungen an die Online-Visualisierung zusammengefasst, für die im darauf folgenden Kapitel ein Konzept erarbeitet wird. Die Umsetzung dieses Konzepts wird in Kapitel 6 beschrieben.

Kapitel 2

Online-Visualisierung

Der Umfang der Daten, die bei numerischen Computersimulationen erzeugt werden, wächst durch immer leistungsfähigere Supercomputer rapide an.
Die Simulationsrechnungen werden üblicherweise in Batchverarbeitung durchgeführt, da sie keine Benutzereingaben erfordern. Das bedeutet, dass die Ergebnisse der Simulation erst nach Beendigung des Jobs betrachtet und nachbereitet werden (Postprocessing). Falsche und unerwünschte Ergebnisse können beim Postprocessing zum Teil erst am Ende der Simulation erkannt werden, so dass unter Umständen sehr viel teure Rechenzeit vergeudet wird. Zudem stellen die Ergebnisse einer Simulation oftmals eine so große Datenmenge dar, dass für deren Analyse eine lokale Workstation aufgrund des begrenzten Speichers nicht ausreicht. Statt dessen müssen auch die Ergebnisse mit einem Supercomputer analysiert werden. Als Alternative hierzu können während der Simulation Zwischendateien erzeugt werden, in denen Teilergebnisse der Simulation festgehalten werden. So wird das Ergebnis der Simulationsrechnung in mehrere Teile zerlegt, die dann zur Analyse auf eine lokale Workstation transportiert werden müssen.

Die Online-Visualisierung hingegen bildet zusammen mit der Simulation eine verteilte Anwendung. Auf die Auslagerung der Ergebnisse in Dateien kann verzichtet werden, da der Datenaustausch über TCP/IP-Sockets stattfindet. Unter Sockets versteht man Kommunikationsendpunkte, die durch eine IP-Adresse und eine Portnummer festgelegt sind. Sie errichten einen direkten Kanal zwischen der Online-Visualisierung und der Simulation.
Die Online-Visualisierung bietet die Möglichkeit, Zwischenergebnisse während der Simulation zu betrachten. Sie dient dabei der Kontrolle des Simulationsprogramms. Die Visualisierung ist nicht als High-End-Analyse zu verstehen, daher ist eine Reduktion der Daten möglich und sinnvoll.

Durch die Online-Visualisierung verbunden mit der interaktiven Steuerung des Simulationsprogramms (Computational Steering) ist der Wissenschaftler in der Lage, die Qualität seiner Simulationsergebnisse zu verbessern und wertvolle Rechenzeit einzusparen. Die Simulation kann in die richtige Richtung gelenkt, oder, falls dies nicht möglich ist, frühzeitig abgebrochen werden [L1]. Außerdem werden jeweils kleinere Datenmengen betrachtet, so dass deren Analyse auf einer lokalen Workstation erfolgen kann.
Zur Realisierung einer Online-Visualisierung wird eine Kopplungsbibliothek benötigt, mit deren Hilfe die Simulation und die Visualisierung verbunden werden können. Außerdem müssen die Daten in Graphiken umgesetzt werden. Zuletzt müssen diese Graphiken in eine graphische Benutzeroberfläche (graphical user interface, GUI) integriert werden (vgl. Abbildung 2.1 auf der nächsten Seite). Im Folgenden werden diese drei Komponenten charakterisiert.

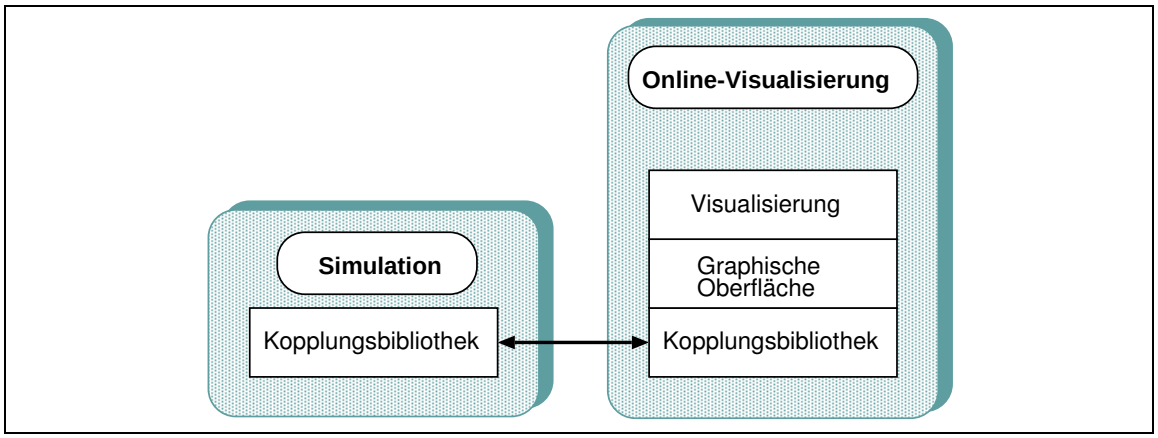


Abbildung 2.1: Komponenten zur Realisierung einer Online-Visualisierung: Simulation und Visualisierung werden durch eine Kopplungsbibliothek miteinander verbunden. Zur Darstellung der Simulationsdaten wird eine Visualisierungssoftware verwendet. Die graphische Oberfläche wird mit Hilfe eines GUI-Toolkits implementiert.

2.1 Die Kopplungsbibliothek VISIT

Um eine gute Grundlage für Steering-Entscheidungen zu bieten, müssen relevante Informationen aus dem laufenden Programm abgegriffen werden. Außerdem muss es möglich sein, Parameter der laufenden Simulation in „Beinahe-Echtzeit“ zu verändern. Langsame Operationen der Visualisierung sollen dabei keinen Einfluss auf die Perfomanz der Simulation haben. Für diese Kommunikation soll die im ZAM[1] entwickelte Kopplungsbibliothek VISIT[2] verwendet werden [L2]. Die Bibliothek liefert Funktionen für den Auf- und Abbau einer Verbindung zwischen einer Simulation und einer Visualisierung, sowie für den transparenten, bidirektionalen Austausch von Daten. Hierbei dient die Visualisierung als Server, der auf Anfragen der Simulation, die hier als Client zu verstehen ist, antwortet. In der Simulation wird eine bestimmte Zeit (timeout) festgelegt, in der Kommunikationsvorgänge abgeschlossen sein müssen. Wird dieses Zeitfenster nicht eingehalten, findet kein Datenaustausch statt. Dadurch wird sichergestellt, dass die Simulation durch Fehler in der Übertragung von Daten und durch Performanzprobleme der Visualisierung nicht beeinträchtigt wird. Als Übertragungsprotokoll dient TCP/IP (siehe Abbildung 2.2 auf der nächsten Seite). Da dieses Protokoll auf allen Plattformen vorhanden ist, ist die Software sehr portabel.
Besteht zwischen der Simulation und der Visualisierung eine Firewall, so muss die Verbindung über einen *ssh-Tunnel* aufgebaut werden. Hierzu wird auf der Visualisierungsseite ein Proxyserver gestartet, so dass die Kommunikation auf der Simulationsseite weiterhin über TCP/IP-Sockets ablaufen kann.

Für den Aufbau einer Verbindung ist es nötig, dass Server und Client sich über einen gemeinsamen Zugangspunkt austauschen. Hierzu wird ein weiterer Server benutzt, der sogenannte Seap[3]-Server. Auf diesem Server hinterlegt die Visualisierung einen Service-Namen und ein Passwort sowie den Hostnamen und die Portnummer. Die Simulation kann diese Informationen abfragen, wenn sie den Service-Namen und das Passwort kennt.

[1] Zentralinstitut für Angewandte Mathematik
[2] **Vis**ualization **I**nterface **T**oolkit
[3] service announcement protocol, Directory-Server

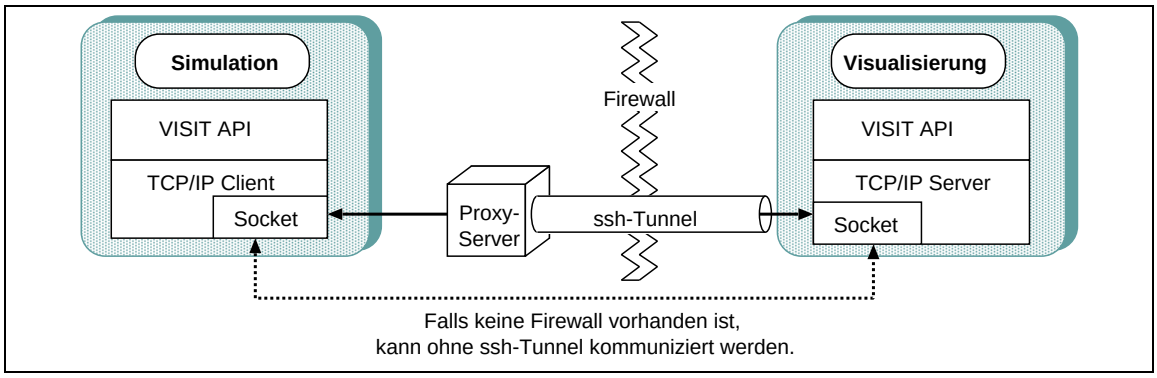


Abbildung 2.2: Organisation einer VISIT-Verbindung: Falls zwischen der Simulation und der Visualisierung eine Firewall existiert, muss die Kommunikation über einen ssh-Tunnel ablaufen. Andernfalls wird direkt über TCP/IP-Sockets kommuniziert.

2.2 Das Visualisierungs-Toolkit (VTK)

Eine weitere Basis für die sinnvolle Einflussnahme auf die laufende Simulation ist die visuelle Präsentation (Visualisierung) der Informationen. Die Aufgabe der Visualisierung ist die Abbildung der Simulationsdaten auf visuell leicht erfassbare Größen wie zum Beispiel Formen und Farben.
Hierfür wird in dieser Arbeit das Visualization Toolkit (VTK) eingesetzt. VTK ist ein Softwaresystem für 3D-Computergraphiken und Visualisierungen. Es baut auf der Funktionalität von OpenGL [L3] auf, bietet aber Funktionen auf einer höheren Abstraktionsebene an. Darüber hinaus bietet VTK eine Vielzahl von Visualisierungsalgorithmen. Im Wesentlichen besteht VTK aus einer C++ Klassenbibliothek und zusätzlichen Schnittstellen zu den Interpretersprachen Tcl/Tk, Java und Python [L4].

In VTK sind zwei verschiedene Modelle umgesetzt: das Graphikmodell und das Visualisierungsmodell [L5].

Das **Graphikmodell** ist eine abstrakte Beschreibung des Aufbaus von 3D-Graphiken. Es kann durch einige wesentliche Objekte beschrieben werden:

Objekt	Funktion
vtkActor	repräsentiert ein beliebiges sichtbares Objekt
vtkLight	produziert eine Lichtquelle
vtkCamera	beinhaltet u.a. die Position und Blickrichtung einer Kamera
vtkMapper	transformiert Daten, die durch andere Vtk-Objekte generiert und modifiziert werden, in Geometrien
vtkRenderer und vtkRenderwindow	stellen die Schnittstelle zwischen dem Fenstersystem und der Graphikbibliothek dar

Graphische Daten werden in Bilder umgesetzt, indem die oben aufgeführten Objekte miteinander verknüpft werden.
Die Portabilität von mit VTK geschriebenen Programmen wird durch sogenannte „device objects“ erreicht. Das sind Objekte, die von abstrakten Klassen abgeleitet sind. Erzeugt der Programmierer eine Instanz der abstrakten Klasse, so werden intern spezifische Klassen, abhängig von dem jeweiligen Fenstersystem oder der Graphikbibliothek, instanziiert.

Das **Visualisierungsmodell** ist ein Datenflussmodell des Renderingprozesses (Visualisierungs-Pipeline), das die Umwandlung von Informationen in graphische Daten beschreibt. Dies geschieht durch die Verbindung von Prozess- und Datenobjekten zu einem Netzwerk.

Prozessobjekte sind Objekte, die Daten entgegennehmen und sie bearbeiten. Sie können anhand der Anzahl ihrer Ein- und Ausgänge unterschieden werden. Solche mit nur einem Ausgang und keinem Eingang heißen Quellen. Sie bilden eine Schnittstelle zu externen Daten oder generieren neue Daten aus lokalen Parametern. Prozessobjekte mit sowohl mindestens einem Ein- als auch Ausgang nennt man Filter. Sie führen Transformationen auf ihren Eingabedaten aus. Objekte mit einem oder mehreren Ein- aber keinen Ausgängen heißen Mapper. Sie beenden die Visualisierungs-Pipeline und setzen die Daten in graphische Primitiven um.
Mapper dienen als Schnittstelle zwischen Visualisierungsmodell und Graphikmodell.
Datenobjekte ermöglichen die Zwischenspeicherung der Daten, die durch das Netzwerk fließen, und erlauben Operationen auf ihnen (vgl. Abbildung 2.3).

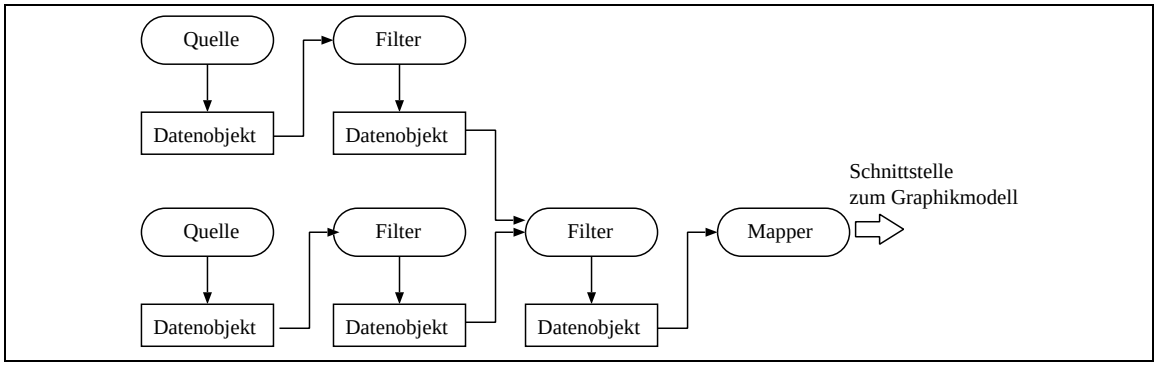


Abbildung 2.3: Die Visualisierungs-Pipeline [L6]: Sie beschreibt das Datenflussmodell der Visualisierung.

Diese beiden Ansätze führen zu einem leichtem Verständnis und einer einfachen Handhabung des Toolkits. Zudem wird eine umfangreiche Palette von 3D-Elementen unterstützt, die für die Visualisierung genutzt werden kann. Der Quellcode von VTK ist frei verfügbar. Dadurch wird eine hohe Transparenz gewährleistet. Des Weiteren ist VTK plattformunabhängig, so dass bei einer Portierung kein zusätzlicher Programmieraufwand entsteht.

2.3 Die graphische Benutzeroberfläche

Die Interaktion zwischen dem Benutzer und der Visualisierung soll durch eine graphische Oberfläche ermöglicht werden. Hierzu wird in dieser Arbeit das GUI-Toolkit Qt eingesetzt. Qt ist eine C++ Klassenbibliothek zur Programmierung von graphischen Oberflächen [L7].

Die Bibliothek stellt einen großen Satz graphischer Objekte zur Interaktion (z.B. Buttons, Menüs und Dialoge), so genannte Widgets[4], zur Verfügung. Mit Hilfe dieser Widgets können Benutzeroberflächen aufgebaut werden. Ein Widget kann mehrere andere Widgets enthalten, die dann innerhalb des Bereichs des übergeordneten Widgets angezeigt werden [L8].
Eine Besonderheit von Qt ist die Art und Weise wie Widgets miteinander kommunizieren: der Signal-Slot-Mechanismus. Die Grundidee besteht darin, dass jedes Widget Signale aussenden kann, ohne dass es wissen muss, welches Objekt auf diese Signale reagiert.
Neben den Signalen gibt es sogenannte Slots. Hierbei handelt es sich um Funktionen, die mit be-

[4] Wortschöpfung aus Window und Gadget (Vorrichtung)

stimmten Signalen verbunden werden können und immer dann ausgeführt werden, wenn ein entsprechendes Signal emittiert wurde. Realisiert wird dieser Mechanismus durch den Meta Object Compiler (moc) von Qt und den C++ Präprozessor [L9].
Der Signal-Slot-Mechanismus setzt auf die Meldungen des zugrunde liegenden Fenstersystems auf. Diese relativ primitiven Meldungen werden an die entsprechenden Widgets weitergeleitet, zum Beispiel das Ereignis, dass eine Maustaste gedrückt wurde, während sich der Zeiger an der Position x,y des Bildschirmes befand. Außer den *Events* des Fenstersystems können auch andere *Events* überwacht werden, zum Beispiel bei der Kommunikation mit anderen Rechnern (*Socket-Events*). Dies spielt bei der Integration von VISIT eine entscheidende Rolle.

Bei der Portierung von Qt-Programmen entsteht kein zusätzlicher Programmieraufwand. Das Programm muss lediglich mit dem entsprechenden Compiler übersetzt und gelinkt werden. Ein weiterer Vorteil von Qt ist die sehr ausführliche Dokumentation, die die Programmierung erleichtert. Für Unix-Systeme gibt es eine Version, die der GPL (General Public Licence) unterliegt und somit frei verfügbar ist, falls man nicht-kommerzielle Software schreiben möchte.

Bisher wurden die Werkzeuge, die für die Online-Visualisierung benötigt werden, allgemein beschrieben. Im Hinblick auf ein konkretes Konzept für die Online-Visualisierung wird im nächsten Kapitel das dieser Arbeit zu Grunde liegende Simulationsprogramm vorgestellt.

Kapitel 3

Simulation von Sternhaufen mit NBODY6++

Die Simulation des dynamischen Verhaltens von Himmelskörpern spielt in der Astrophysik eine große Rolle. Sie trägt zum Verständnis von Struktur und Entwicklung von Planetensystemen, Sternhaufen und Galaxien bei.

In dieser Arbeit werden Simulationen von Kugelsternhaufen, die mit dem Simulationsprogramm NBODY6++ durchgeführt werden, betrachtet. Vor allem gravitative Wechselwirkungen spielen bei der Simulation der Sterne eine große Rolle. Es handelt sich um ein typisches N-Körper-Problem, da alle Sterne miteinander wechselwirken. Weil aber meist eine enorme Zahl von Sternen untersucht werden soll (bei Sternhaufen 10^4 bis 10^6 Sterne), stellt die Berechnung der Flugbahnen und Kräfte eine große Herausforderung an die Hard- und Software dar. Daher muss ein Kompromiss zwischen einer besonders realistischen Modellbildung und einer effizienten Bearbeitung gefunden werden.

Einen guten Ansatz, beiden Anforderungen gerecht zu werden, bietet die Methode der direkten Integration der Newtonschen Bewegungsgleichung.

$$a_i \quad = \quad -G \sum_{j=1, j\neq i}^{N} \frac{m_j (r_i - r_j)}{|r_i - r_j|^3} \tag{3.1}$$

Newtonsche Bewegungsgleichung: m_i ist die Masse, r_i die Position und a_i ist die Beschleunigung des i-ten Partikels. G und N sind die Gravitationskonstante und die Anzahl der Partikel. Die Gleichung beschreibt die Bewegung eines Partikels aufgrund der Wechselwirkungen mit allen anderen Partikeln

Die Lösung dieser Differentialgleichung ist für mehr als zwei Partikel nicht analytisch zu bestimmen. Daher werden numerische Verfahren eingesetzt, um eine Näherungslösung zu finden.
Die numerische Integration der Newtonschen Bewegungsgleichung wurde in dem Simulationsprogramm NBODY6 (Aarseth, 1999) umgesetzt und durch die Anwendung verschiedener Verfahren optimiert. Dieses Programm wurde für den Einsatz auf Supercomputern zusätzlich parallelisiert (NBODY6++, Spurzem, 1999)[L10].

Es tragen im Wesentlichen die folgenden Verfahren zur Effizienz von NBODY6++ bei:

- das Hermitesche Verfahren,
- die Verwendung von Blockzeitschritten,

- die Regularisierung (Zusammenfassung) von Partikeln, die sich sehr nahe kommen und
- das Ahmad-Cohen Verfahren.

Diese Komponenten werden in den nächsten Unterkapiteln genauer erläutert.

3.1 Das Hermitesche Verfahren

Das Hermitesche Verfahren ist ein numerisches Integrationsverfahren – genauer ein Prädiktor-Korrektor-Verfahren. Bei diesen Verfahren wird zunächst mit einem expliziten Integrationsverfahren niedriger Ordnung (Prädiktor) eine Näherung bestimmt. Diese Näherung wird dann mit einem impliziten Verfahren höherer Ordnung (Korrektor) verbessert.
NBODY6++ benutzt dieses Verfahren um die Positionen und Geschwindigkeiten der Sterne zu berechnen. Für die erste Näherung wird eine Taylor-Entwicklung um einen Startzeitpunkt t_0, zu dem die Orte, Geschwindigkeiten und Beschleunigungen eines Partikels bekannt sind, herangezogen:

$$r_{p,i}(t) = r_{0,i} + v_{0,i}(t-t_0) + a_{0,i}\frac{(t-t_0)^2}{2} + \dot{a}_{0,i}\frac{(t-t_0)^3}{6} \tag{3.2}$$

$$v_{p,i}(t) = v_{0,i} + a_{0,i}(t-t_0) + \dot{a}_{0,i}\frac{(t-t_0)^2}{2} \tag{3.3}$$

Dabei werden a_0 und $\dot{a}_0$ über die Newtonsche Bewegungsgleichung und deren ersten Ableitung bestimmt:

$$a_{0,i} = -G\sum_{j=1,j\neq i}^{N}\frac{m_j(r_i-r_j)}{|r_i-r_j|^3} \tag{3.4}$$

$$\dot{a}_{0,i} = -G\sum_{j=1,j\neq i}^{N} m_j\left(\frac{v_{0,i}-v_{0,j}}{|r_{0,i}-r_{0,j}|^3} + \frac{3(r_{0,i}-r_{0,j})\left((v_{0,i}-v_{0,j})(r_{0,i}-r_{0,j})\right)}{|r_{0,i}-r_{0,j}|^5}\right) \tag{3.5}$$

Die Genauigkeit dieser Näherung ist jedoch für die Simulation nicht ausreichend. Der Fehler, der dabei auftritt, würde selbst bei sehr kleinen Schrittweiten $(t-t_0)$ bald untragbar werden.
Daher wird ein Korrektor verwendet, der die mit dem Prädiktor bestimmte Näherung zum Zeitpunkt t verbessert. Der Korrektor basiert ebenfalls auf einer Taylor-Entwicklung, die jedoch um zwei weitere Summanden erweitert wurde:

$$r_1(t) = r_p + a_0^{(2)}\frac{(t-t_0)^4}{24} + a_0^{(3)}\frac{(t-t_0)^5}{120} \tag{3.6}$$

$$v_1(t) = v_p + a_0^{(2)}\frac{(t-t_0)^3}{6} + a_0^{(3)}\frac{(t-t_0)^4}{24} \tag{3.7}$$

In diesen Gleichungen sind die zweite und dritte Ableitung der Beschleunigung ($a_0^{(2)}$ und $a_0^{(3)}$) unbekannt. Sie werden durch die Hermitesche Interpolation angenähert. Hierzu wird ebenfalls eine Taylor-Entwicklung durchgeführt, diesmal jedoch für die Beschleunigung a und deren Ableitung $\dot{a}$:

$$a_1(t) = a_0 + \dot{a}_0(t-t_0) + a_0^{(2)}\frac{(t-t_0)^2}{2} + a_0^{(3)}\frac{(t-t_0)^3}{6} \tag{3.8}$$

$$\dot{a}_1(t) = \dot{a}_0 + a_0^{(2)}(t-t_0) + a_0^{(3)}\frac{(t-t_0)^2}{2} \tag{3.9}$$

Die Gleichungen werden nach $a_0^{(2)}$ und $a_0^{(3)}$ aufgelöst:

$$a_0^{(3)} = 12\frac{a_0 - a_1}{(t-t_0)^3} + 6\frac{\dot{a}_0 + \dot{a}_1}{(t-t_0)^2} \tag{3.10}$$

$$a_0^{(2)} = -6\frac{a_0 - a_1}{(t-t_0)^2} - 2\frac{2\dot{a}_0 + \dot{a}_1}{t-t_0} \tag{3.11}$$

a_1 und $\dot{a}_1$ können mit Hilfe der Newtonschen Bewegungsgleichung angenähert werden, indem die Werte aus dem Prädiktorschritt in eingesetzt werden. Zusammen mit den Gleichungen (3.6) und (3.7) ergibt sich hieraus das Hermitesche Integrationsverfahren [L11].

3.2 Die Verwendung von Blockzeitschritten

Die Schrittweite ($\Delta t = t_{i+1} - t_i$), die bei dem im vorigen Kapitel beschriebenen Verfahren angewendet wird, ist entscheidend für den Fehler. Um diesen möglichst gering zu halten, müsste die Schrittweite durch die beiden Partikel mit dem kleinsten Abstand vorgegeben werden, da ihre Beschleunigung sich durch die Gravitation am stärksten ändert. Obwohl sich die Beschleunigung anderer Partikel weniger stark ändert, würden die Positionen und Geschwindigkeiten dieser Partikel ebenfalls in diesen sehr kleinen Zeitabständen berechnet. Da dieser Vorgang jedoch sehr zeitaufwendig ist, wird jedem Partikel ein individueller Zeitschritt zugeordnet. So können die Daten der Partikel, die sich sehr langsam bewegen, in größeren Zeitabständen neu berechnet werden. Zur Berechnung der Partikeldaten zu einem bestimmten Zeitpunkt werden jedoch die Daten aller Partikel zum vorhergehenden Zeitpunkt benötigt. Diese werden, falls sie nicht vorliegen, durch den im Kapitel 3.1 beschriebenen Prädiktor angenähert. Um den Zeitschritt eines Partikels i zu bestimmen, liefert Aarseth folgende Formel:

$$\Delta t_i = \sqrt{\eta \frac{|a_{1,i}||a_{1,i}^{(2)}| + |\dot{a}_{1,i}|^2}{|\dot{a}_{1,i}||a_{1,i}^{(3)}| + |a_{1,i}^{(2)}|^2}} \tag{3.12}$$

wobei durch die Wahl von η der Fehler kontrolliert werden kann [L12].

Werden die exakten Zeitschritte Δt_i verwendet, so wird pro Zeitschritt nur ein Partikel neu berechnet. Um dies zu vermeiden, werden die Zeitschritte auf Potenzen von 2 diskretisiert. Ein Partikel erhält ausgehend von dem mit Formel (3.12) berechneten exakten Zeitschritt den nächst niedrigeren diskreten Zeitschritt. Auf diese Weise erhalten ganze Gruppen (Blöcke) von Partikeln den gleichen Zeitschritt und können gleichzeitig berechnet werden (vgl. Abbildung 3.1 auf der nächsten Seite).

Der Zeitschritt eines Partikels ist nicht konstant, er wird am Ende jeder Iteration neu berechnet.

3.3 KS-Regularisierung

Im Zentrum eines Sternhaufens kann es vorkommen, dass Sterne sehr nahe zusammenkommen. Bei sehr kleinen relativen Abständen treten jedoch starke numerische Probleme (Singularitäten) auf. Um diese zu vermeiden, wird die Kustaanheimo-Stiefel-Regularisierung (KS-Regularisierung) verwendet. Bei diesem Verfahren werden jene Sterne durch ihr gemeinsames Massenzentrum ersetzt. Mit diesem zusammengesetzten Partikel wird dann die Integration durchgeführt. Um die Bahn des KS-Partikels zu bestimmen, wird eine Koordinatentransformation des Raumes und der Geschwindigkeiten mittels einer regulären 4x4 Matrix durchgeführt. Eine detailliertere Beschreibung des Verfahrens ist in [L12] beschrieben.

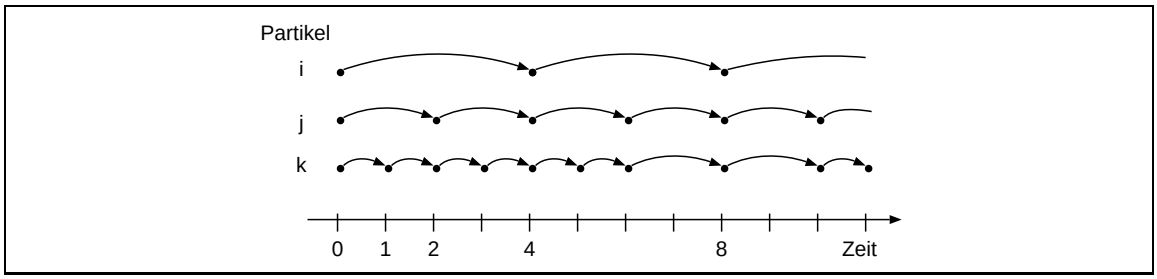


Abbildung 3.1: Blockzeitschritte: Diese Abbildung zeigt anhand der Beispielpartikel i, j und k die Funktionsweise von Blockzeitschritten. Zum Zeitpunkt 0 sind die Daten aller Partikel durch eine Startverteilung vorgegeben. Partikel k hat zu Anfang die kleinsten Zeitschritte. Partikel j und i haben doppelte bzw. vierfache Schrittweiten. Die Schrittweite wird nach jeder Iteration justiert. Zum Zeitpunkt 4 müssen alle drei Partikel neu berechnet werden, während zum Zeitpunkt 2 nur die Partikel j und k komplett neu berechnet werden.

3.4 Das Ahmad-Cohen Verfahren

Die Simulation von Kugelsternhaufen nimmt durch die hohe Anzahl von Sternen, die miteinander wechselwirken, sehr viel Rechenzeit in Anspruch. Bei jeder Iteration des Hermiteschen Verfahrens müssen sämtliche Kräfte, die von allen anderen Sternen auf einen einzelnen Stern wirken, aufsummiert werden. Das Ahmad-Cohen Nachbarschema ist ein Verfahren, das den Aufwand für die Berechnung der Kräfte reduziert [L10].
Die Kraft, die von allen anderen Sternen auf einen Stern wirkt, wird hierzu in zwei Komponenten ($a_{i,reg}$ und $a_{i,irr}$) aufgeteilt. $a_{i,irr}$ bezeichnet dabei den irregulären Anteil der Kräfte, die von Sternen in einer bestimmten Nachbarschaft ausgehen. Dieser Anteil unterliegt größeren Schwankungen. $a_{i,reg}$ dagegen ist die Summe der regulären Kräfte, die von den weiter entfernten Sternen ausgehen. Es werden zwei Zeitskalen ($t_{i,irr}$ und $t_{i,reg}$) eingeführt, die festlegen, wann $a_{i,irr}$ und $a_{i,reg}$ mit welcher Genauigkeit berechnet werden (vgl. Abbildung 3.2). Zu den Zeitpunkten $t_{i,irr}$ wird der Korrektor aus dem Hermiteschen Verfahren (vgl. Kapitel 3.1) lediglich für $a_{i,irr}$ angewendet. $a_{i,reg}$ wird nur durch den (weniger zeitaufwendigen) Prediktor angenähert. Zu den Zeitpunkten $t_{i,reg}$ wird für beide Anteile der Korrektor verwendet.

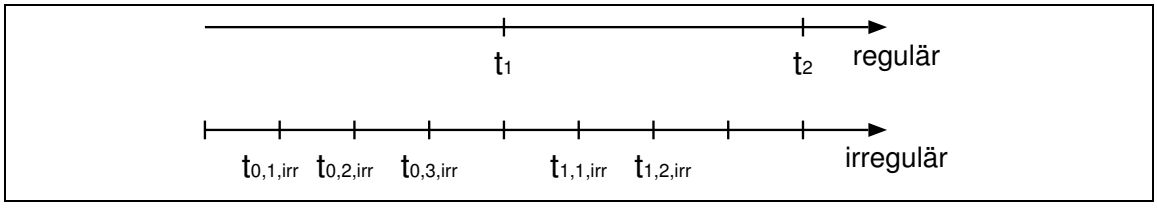


Abbildung 3.2: Unterschiedliche Zeitskalen im Ahmad-Cohen Verfahren: Der irreguläre Anteil der Kräfte, die durch die Sterne einer bestimmten Nachbarschaft auf den betrachteten Stern ausgeübt werden, muss häufiger bestimmt werden, als der Anteil der regulären Kräfte.

Realisiert wird das Verfahren durch Nachbarschaftslisten, die immer dann aktualisiert werden, wenn auch die regulären Kräfte genau bestimmt werden.

Aus den oben beschriebenen Eigenschaften der Simulation ergibt sich, dass die Ergebnisse der Simulation einige Besonderheiten aufweisen, die bei der Visualisierung berücksichtigt werden müssen:

- Durch die Verwendung von Blockzeitschritten liegen nicht zu allen Sternen Daten zu einem Simulationszeitpunkt vor. Trotzdem sollen in der Visualisierung alle Sterne dargestellt werden.
- Bei der Darstellung von Flugbahnen müssen eventuelle KS-Regularisierungen beachtet werden. Auch mehrfach regularisierte Partikel können vorkommen.

Im folgenden Kapitel werden die Anforderungen an die Visualisierung ausführlich beschrieben.

Kapitel 4

Anforderungen an die Online-Visualisierung

In dieser Arbeit wird exemplarisch anhand der zuvor vorgestellten Anwendung „Simulation von Sternhaufen“ ein Konzept für die Integration der VISIT-Bibliothek in die nicht-kommerzielle Graphiksoftware VTK entwickelt und mit Hilfe des GUI-Tools Qt eine Oberfläche zur Visualisierung von Simulationsdaten und zur Steuerung des Simulationsprogramms NBODY6++ implementiert werden. Die Anforderungen an eine solche Online-Visualisierung können in folgende Teilbereiche gegliedert werden, die in diesem Kapitel weiter ausgeführt werden:

- Zwischenspeicherung der Simulationsdaten,
- Integration von VISIT,
- Steering,
- Anforderungen an die Visualisierung und
- Entwurf einer graphischen Benutzeroberfläche.

4.1 Zwischenspeicherung der Simulationsdaten

Für die Darstellung der Flugbahnen der Sterne werden nicht nur die aktuelle Position, sondern auch vorhergehende Positionen benötigt. Dazu muss auf der Visualisierungsseite eine Zwischenspeicherung der Daten möglich sein.
Die Funktionalität des Zwischenspeichers kann in zwei Komponenten gegliedert werden:

- die Verwaltung des Speicherplatzes und
- die Verwaltung der Simulationsdaten.

Die Verwaltung des Speicherplatzes umfasst Funktionen zur (Re-) Initialisierung des Zwischenspeichers, Speicherallokation und -freigabe sowie für Statusabfragen.
Zur Verwaltung der Simulationsdaten werden die folgenden Funktionen benötigt:
Die Simulationsdaten werden in Blöcken variabler Größe an die Visualisierung geschickt und dann in dem Zwischenspeicher abgelegt. Außerdem werden Funktionen zur Abfrage eines Datensatzes zu einem gewünschten Zeitpunkt sowie zur Ermittlung von Flugbahnen einzelner Sterne bereitgestellt. Durch die Verwendung der in Kapitel 3.2 auf Seite 11 beschriebenen Blockzeitschritte liegen nicht zu jedem Zeitpunkt die Daten aller Sterne vor. Daher muss die Position dieser Sterne mit Hilfe einer Vorausberechnung angenähert werden. Auch das ist eine Aufgabe des Zwischenspeichers.

Der Zwischenspeicher wird in einem eigenständigen Software-Modul entwickelt. So kann er für ähnliche Anwendungen wiederverwendet werden.

4.2 Integration von VISIT

Um die Simulationsdaten in der Visualisierung verfügbar zu machen, werden Simulation und Visualisierung miteinander gekoppelt. Hierzu wird die Kopplungsbibliothek VISIT eingesetzt, die auf Socket-Kommunikation basiert. Die Visualisierung übernimmt hierbei die Funktion eines Servers und muss ständig bereit sein, die Anfragen der Simulation zu beantworten. Gleichzeitig muss jedoch eine Interaktion mit dem Benutzer möglich sein.
Zur Realisierung des quasi-parallelen Ablaufs der Kommunikation zwischen der Simulation und der Visualisierung sowie zwischen dem Benutzer und der Visualisierung können verschiedene Methoden angewendet werden.
Eine davon ist die Verwendung von zwei nebenläufigen Threads, wobei ein Thread nur dafür zuständig ist, auf Socketaktivitäten zu warten und die Anfragen der Simulation zu beantworten. während der andere die Benutzereingaben beantwortet. Diese Lösung ist jedoch recht fehleranfällig, da beide Threads auf denselben Speicher zugreifen müssen. Daher muss bei der Verwendung von Threads darauf geachtet werden, dass jeweils nur ein Thread zu einem Zeitpunkt Zugriff auf den Speicher hat.
Eine weitere Methode stellt die Verwendung von *Events* dar. Wie in Kapitel 2.3 schon erwähnt wurde, können *Socket-Events* von Qt überwacht werden. Diese werden neben den Events des Fenstersystems, die durch die Benutzerinteraktion hervorgerufen werden, in einer Warteschlange (*eventqueue*) eingetragen und nacheinander bearbeitet. So muss nicht explizit auf Socketaktivitäten gewartet werden, und es kann weiter auf Benutzereingaben reagiert werden. In dieser Arbeit wurde die letztere Methode verwendet, da sie weniger fehleranfällig ist.

4.3 Steuerung der Simulation (Steering)

Die Kopplungsbibliothek VISIT ermöglicht eine bidirektionale Kommunikation zwischen Simulation und Visualisierung. Neben der Online-Visualisierung, bei der Daten von der Simulation zur Visualisierung übertragen werden, können auch Daten von der Visualisierung zur Simulation übertragen werden. Über diesen Weg kann die Simulation von der Visualisierung aus gesteuert werden. Auf der Simulationsseite werden Simulationsdaten in einem Zwischenspeicher gesammelt, um dann als größere Blöcke an die Visualisierung geschickt zu werden. Durch das Steering können die Parameter des Zwischenspeichers der Simulation beeinflusst werden. So kann zum Beispiel die Größe der Blöcke vorgegeben werden.
Ausserdem ist es möglich, den Programmablauf von NBODY6++ zu steuern und Einfluss auf das Ein- und Ausgabeverhalten der Simulation zu nehmen.

4.4 Anforderungen an die Darstellung der Simulationsdaten

Durch die Visualisierung werden dem Benutzer die Simulationsdaten in einer leicht verständlichen Form präsentiert. Höherdimensionale Daten werden auf 2- bzw. 3-dimensionale Darstellungen abgebildet.
Dabei dient die Visualisierung wie im Kapitel 2 beschrieben der Kontrolle des Simulationsprogramms. Die Hauptintention ist also nicht die Darstellung der Simulationsdaten im Sinne einer High-End-Visualisierung, sondern die Darstellung in einer vertretbaren Geschwindigkeit. Diese richtet sich nach der Aktualisierungsrate der Simulation.

3-dimensionale Darstellung:

Die 3-dimensionale Darstellung verdeutlicht die Positionen der Sterne im Raum. Als weitere Hilfsmittel hierfür werden optional die *Bounding-Box*[1] sowie 3-dimensionale Achsen dargestellt.
Weiterhin werden die Geschwindigkeiten der Sterne durch Pfeile sichtbar gemacht. Andere Attribute, aus denen der Benutzer selbst wählen kann, sollen anhand verschiedener Farben und Größen der die Sterne repräsentierenden Objekte dargestellt werden. Optional können auch die Flugbahnen der einzelnen Sterne angezeigt werden. Auch hier wählt der Benutzer ein Attribut aus, auf welches die Farben der Flugbahnen abgebildet werden.
Bei der Farbgebung der Flugbahnen macht es Sinn, als Grundlage die Wertemenge des betrachteten Attributs im gesamten Speicher festzulegen, da theoretisch alle Werte hieraus angenommen werden können. Die Objekte, die die Sterne repräsentieren, erhalten ihre Farben bezogen auf die Wertemenge des Attributs zum aktuell betrachteten Simulationszeitpunktes. Dadurch kann es vorkommen, dass die Flugbahn, obwohl sie nach dem selben Attribut eingefärbt wurde, an der aktuellen Position des Sternes eine andere Farbe hat als der Stern selbst. Wird dagegen auch bei den Flugbahnen die Wertemenge zu dem aktuellen Simulationszeitpunkt als Grundlage der Farbverteilung bestimmt, können Attribute, die außerhalb dieses Wertebereichs liegen, nicht korrekt dargestellt werden. Da beide Möglichkeiten sowohl Vor- als auch Nachteile haben, wird dem Benutzer die Möglichkeit gegeben, aus den beiden oben beschriebenen Darstellungsarten zu wählen. Alternativ dazu kann der Benutzer auch selbst einen anderen Wertebereich angeben.

2-dimensionale Darstellung:

Bei der 2-dimensionalen Darstellung werden Orts- und Geschwindigkeitskomponenten gegeneinander abgebildet. Der Benutzer selbst wählt die für ihn relevante Kombination aus. Die Ort-Geschwindigkeits-Graphik (*Phasenraum*) beschreibt den Bewegungszustand des Systems. Ändert sich die Geschwindigkeit eines Partikels, so verändert es seine Position im Phasenraum. An der Darstellung des Phasenraumes kann man ablesen, wie sich das gesamte System verhält.

4.5 Entwurf einer graphischen Benutzeroberfläche

Die 2- und 3-dimensionalen Darstellungen der Simulationsdaten werden in eine mit Qt generierte graphische Oberfläche integriert. Zur Steuerung dieser Darstellungen werden außerdem Werkzeuge wie Menüs, Eingabefelder, Schalter, usw. zur Verfügung gestellt. Mit Hilfe dieser Werkzeuge können die Eigenschaften der Objekte, die die Simulationsdaten darstellen, beeinflußt werden. Zum Beispiel können Attribute festgelegt werden, auf die die Farben der Objekte abgebildet werden sollen.
Auch die Steuerung der Simulation wird über Eingabefelder in der graphischen Oberfläche realisiert. Des Weiteren beinhaltet die Benutzeroberfläche Statusanzeigen, die Aufschluss über den Zustand des Zwischenspeichers und der Verbindung mit der Simulation geben.

Die oben beschriebenen Anforderungen werden im nächsten Kapitel nochmals aufgegriffen. Hier wird ein Konzept entwickelt, das auf diese Forderungen zugeschnitten ist und anhand dessen die Online-Visualisierung realisiert wird.

[1] kleinster Quader, der alle Sterne umschließt

Kapitel 5

Konzept

In diesem Kapitel wird ein Konzept für die Online-Visualisierung entwickelt. Um auf die im vorigen Kapitel aufgeführten Anfordungen einzugehen, werden zunächst die Simulationsdaten analysiert, da sie die Grundlage für alle weiteren Überlegungen bilden. In den Kapiteln 5.2 bis 5.7 werden dann die Module entworfen, durch die die Online-Visualisierung (*xnbody*) realisiert wird. Anschließend wird ein Modul vorgestellt, welches die Datensätze eines Simulationszeitpunktes für die Visualisierung bereitstellt. Um die Daten zwischen der Simulation und der Visualisierung zu transportieren, wird die Kopplungsbibliothek VISIT in die Online-Visualisierung integriert. Abschließend werden die Konzepte für die Visualisierung und die graphische Benutzeroberfläche vorgestellt.

5.1 Datenanalyse

Die Simulationsdaten werden in einem Zwischenspeicher auf der Simulationsseite in chronologischer Reihenfolge gesammelt. Sie werden dann in Blöcken bestimmter Größe an die Visualisierung geschickt. Die Daten haben die Struktur eines 2-dimensionalen Arrays von Fließkommazahlen (vgl. Abbildung 5.1).

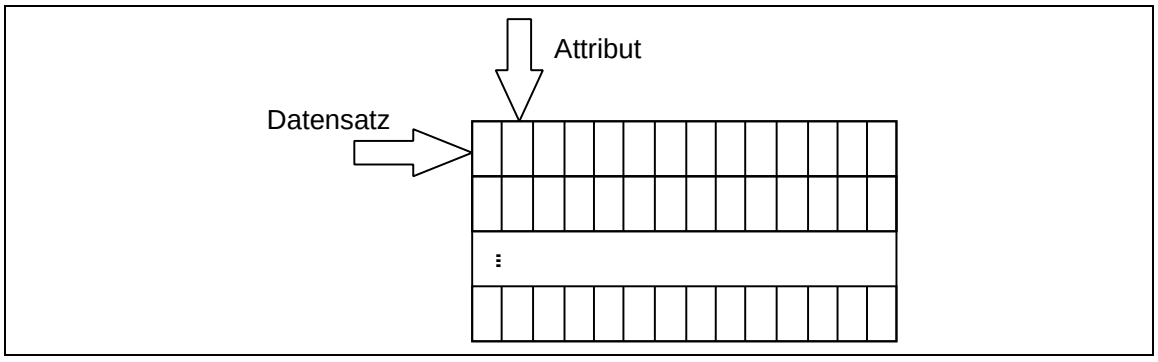


Abbildung 5.1: Struktur der Simulationsdaten: Die Daten bestehen aus einem Feld von Datensätzen. Diese enthalten eine feste Anzahl von Attributen.

Das Feld besteht aus mehreren Datensätzen mit jeweils einer festen Anzahl von Attributen. Derzeit wird nur ein Teil der in der Simulation berechneten Attribute eines Partikels betrachtet, nämlich diejenigen, die für die Vorausberechnung der Positionen und Geschwindigkeiten (vgl. 5.2.2 auf Seite 22) benötigt werden. Das Konzept ist jedoch so angelegt, dass später weitere Attribute berücksichtigt werden können.
Die Datensätze können in unterschiedliche Typen eingeteilt werden. Zur Unterscheidung der einzelnen Typen enthalten alle Datensätze ein Attribut, das den Typ des Datensatzes angibt. Dieses

Attribut legt fest, wie die anderen Attribute des Datensatzes interpretiert werden müssen (vgl. Abbildung 5.2).

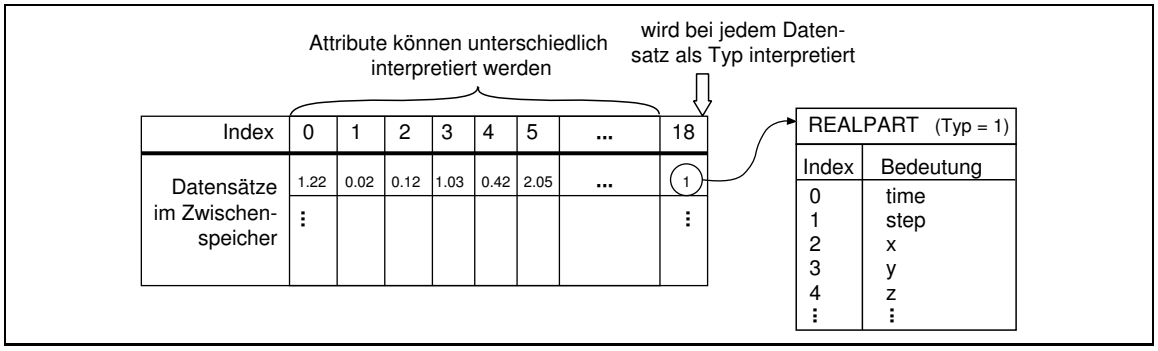


Abbildung 5.2: Interpretation der Simulationsdaten: An einem *REALPART*-Datensatz wird exemplarisch demonstriert, wie die Daten interpretiert werden.

Unter den Datensätzen befinden sich neben den Partikeldaten auch Meta-Datensätze. Diese werden bei der Visualisierung nicht dargestellt, enthalten aber Informationen, die für die Interpretation der Partikeldatensätze wichtig sind.

Es werden zunächst die Partikeldatensätze erläutert. Diese können die Typen ***REALPART*** und ***KSPART*** haben. Durch Datensätze mit diesen Typen werden die Sterne charakterisiert. Sie machen den Hauptteil der Simulationsdaten aus. Partikeldatensätze enthalten die folgenden Attribute:

Attribut	Bedeutung
T0	Simulationszeitpunkt
STEP	Schrittweite, die angibt, in welcher Häufigkeit die Daten des Sterns aktualisiert werden
X0, X1, X2	Ortsvektor
X_DOT0, X_DOT1, X_DOT2	Geschwindigkeitsvektor
F0, F1, F2	Kraft
F_DOT0, F_DOT1, F_DOT2	Änderung der Kraft
NAME	Bezeichnung des Sterns im Simulationsprogramm
RHO	Dichte
I	eindeutige Identifikationsnummer
TYPE	Typ des Datensatzes, in diesem Fall *REALPART* bzw. *KSPART*

Meta-Datensätze können wie folgt charakterisiert werden:

- ***KSREG*: Informationen über eine Regularisierung**

  Hier sind Informationen über den Zusammenschluss zweier Partikel zu finden wie zum Beispiel die Identifikationsnummern der beiden Partikel, der Zeitpunkt, zu dem sie zusammengefasst wurden und die neue Identifikationsnummer, unter der das regularisierte Partikel zu finden ist.

- ***KSTERM*: Informationen über die Auflösung einer Regularisierung**

  Wenn die Regularisierung zweier Partikel wieder aufgehoben wird, wird dieses Ereignis in einem Datensatz diesen Typs festgehalten. Es werden die Nummern der beteiligten Partikel sowie der Zeitpunkt der Auflösung angegeben.

- ***STAT*: Status**

  Dieser Datensatz gibt Aufschluss über die Anzahl der simulierten Partikel sowie über die Anzahl der KS-regularisierten Partikel. Zu Anfang der Visualisierung muss ein Datensatz des Typs *STAT* übertragen werden, damit in der Visualisierung entsprechende Initialisierungen vorgenommen werden können.

5.2 Buffering-Mechanismus

Sowohl auf der Simulations- als auch auf der Visualisierungsseite werden Zwischenspeicher benötigt. Auf der Simulationsseite werden die Daten in dem Speicher gesammelt und dann als größere Datenpakete an die Visualisierung geschickt. So wird der Aufwand für die Kommunikation auf der Simulationsseite möglichst gering gehalten.
Auf der Visualisierungsseite werden die Daten, die für die Visualisierung notwendig sind, in einem weiteren Zwischenspeicher aufgenommen. Die Funktionalität des Zwischenspeichers gliedert sich in die Verwaltung des Speicherplatzes und die Verwaltung der Daten (vgl. Abbildung 5.3).

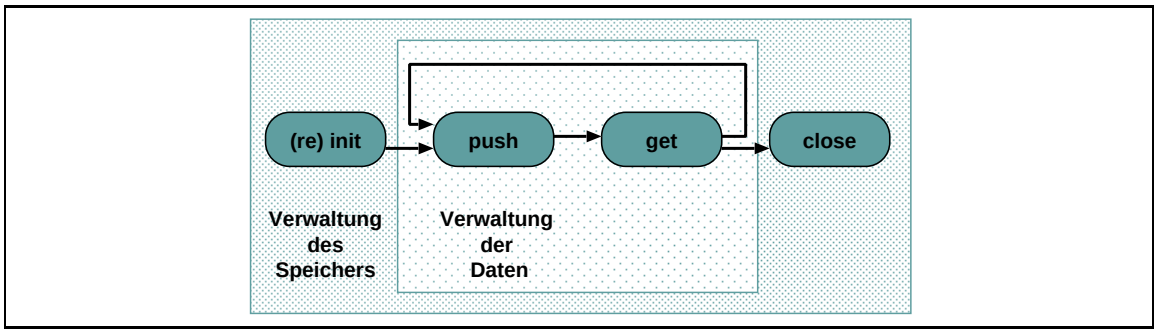


Abbildung 5.3: Ablaufdiagramm zur Verwaltung des Zwischenspeichers: Nach der Initialisierung des Speichers werden im Programmverlauf mehrfach Daten ein und ausgelesen. Zum Schluss wird der Zwischenspeicher geschlossen.

Durch die Verwaltung des Speicherplatzes wird unter anderem die Struktur des Zwischenspeichers festgelegt. Da die Simulationsdaten in chronologischer Reihenfolge an die Visualisierung geschickt werden, ist es in diesem Fall sinnvoll, einen Zwischenspeicher mit FIFO[1]-Verhalten zu implementieren. Das bedeutet, wenn in dem vollständig belegten Speicher neue Daten abgelegt werden sollen, werden zuerst die Daten mit der frühesten Simulationszeit aus dem Speicher entfernt. Dies gewährleistet, dass die Reihenfolge der Daten erhalten bleibt. Der Zwischenspeicher enthält somit einen Ausschnitt der Simulationsdaten über einen gewissen Zeitraum.
Neben der Realisierung der Ersetzungsstrategie umfasst die Verwaltung des Speichers auch Funktionen zur Initialisierung und Auflösung des Zwischenspeichers.

Für die Verwaltung der Daten werden sowohl eine schreibende als auch lesende Funktionen benötigt. In den folgenden Unterkapiteln werden diese Funktionen detailliert erläutert.

[1] First In, First Out

5.2.1 Speicherung der Daten

Die Daten treffen auf der Visualisierungsseite in Blöcken ein. Bei der Speicherung ergibt sich dadurch eine der folgenden vier Situationen:

1. im Speicher ist ausreichend Platz vorhanden für den zu speichernden Datenblock,
2. der Speicher ist zwar nicht voll, der freie Platz reicht aber nicht aus,
3. der Speicher ist vollständig belegt,
4. der Speicher ist grundsätzlich zu klein für die Menge der Datensätze.

Ist genügend Platz im Speicher vorhanden (Fall 1), können die Daten direkt hinter den zuletzt gespeicherten Daten abgelegt werden.
Der 3. Fall ist ein Sonderfall des 2. Falles. Im Folgenden werden diese beiden Fälle zusammen betrachtet. In beiden Fällen ist eine Ersetzungsstrategie erforderlich. Wie oben beschrieben wurde, wird hier die FIFO-Ersetzungsstrategie gewählt.
Tritt der 4. Fall ein, können nicht alle Datensätze aufgenommen werden. Auch hier wird dann die FIFO-Strategie angewendet. Das bedeutet, dass das ankommende Paket aufgeteilt wird und die Datensätze mit den frühesten Simulationszeitpunkten nicht gespeichert werden.

Werden zwei Partikel in der Simulation zusammengefasst, so wird ein Datensatz des Typs *KSREG* erzeugt. Es entsteht ein neues Partikel, das dann mit einer neuen Identifikationsnummer versehen und mit dem Typ *KSPART* gekennzeichnet wird. Die Auflösung einer KS-Regularisierung wird mit einem Datensatz des Typs *KSTERM* dokumentiert. Durch das Ersetzen von alten Datensätzen kann es vorkommen, dass Informationen über diesen Vorgang verloren gehen und die Flugbahnen von KS-regularisierten Partikeln eventuell nicht vollständig verfolgt werden können. Für die Ermittlung der Flugbahnen sind der Zeitpunkt der Regularisierung und der Auflösung und die Nummern der beteiligten Partikel relevant. Diese Daten werden daher auf jeden Fall separat registriert.

5.2.2 Abfrage von Datensätzen

In Kapitel 4.4 wurde beschrieben, dass die Simulationsdaten auf unterschiedliche Arten visualisiert werden können. Zum einen können die Positionen und andere Attribute der Sterne zu einem Zeitpunkt dargestellt werden. Zum anderen können aber auch Flugbahnen betrachtet werden. Diese beiden Möglichkeiten setzen zwei unterschiedliche Datenabfragen voraus. Die Abfrage von Datensätzen kann entweder nach einem Simulationszeitpunkt oder bestimmten Partikeln erfolgen (vgl. Abbildung 5.4 auf der nächsten Seite).

Abfrage nach einem Simulationszeitpunkt

Wird nach einem bestimmten Simulationszeitpunkt abgefragt, so wird der nächst frühere Zeitpunkt im Speicher gesucht, an dem eine Partikelposition zu finden ist. Da aufgrund der Verwendung von Blockzeitschritten (vgl. Kapitel 3.2) nicht zu jedem Zeitpunkt alle Daten vorliegen, werden mit Hilfe einer Vorausberechnung die Positionen und Geschwindigkeiten der betreffenden Partikel errechnet.

Abfrage nach einzelnen Sternen

Die Abfrage nach Partikeln bietet die Möglichkeit, Flugbahnen darzustellen. Hierbei wird der gesamte Zwischenspeicher nach allen Datensätzen mit der entsprechenden Identifikationsnummer

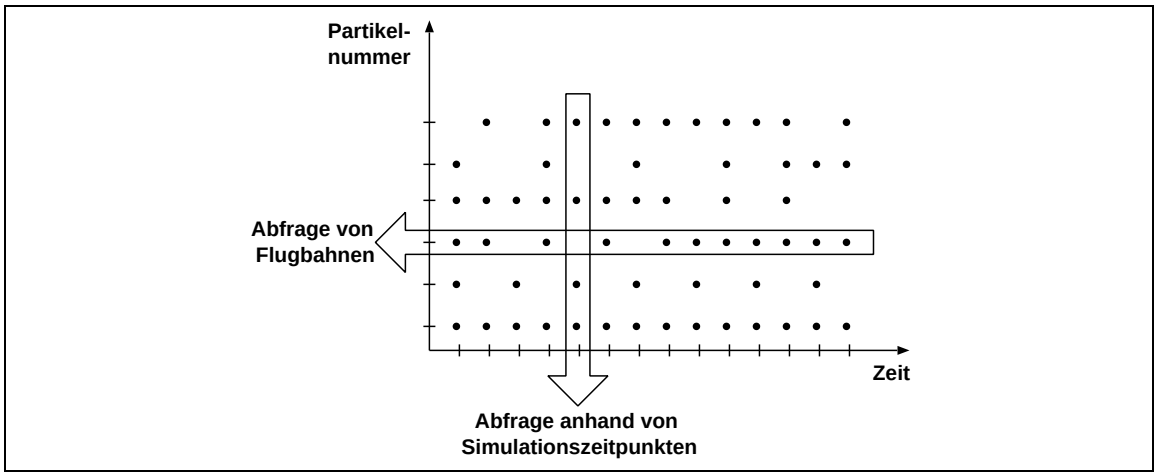


Abbildung 5.4: Abfrage der Daten im Zwischenspeicher: Es können zwei verschiedene Kriterien bei der Abfrage angegeben werden. Zum einen kann ein Simulationszeitpunkt angegeben werden. Dabei kommt es vor, dass der Zwischenspeicher zu einigen Partikeln keine Daten zu dem gewünschten Zeitpunkt enthält. Diese werden dann durch eine Vorausberechnung angenähert. Zum anderen können Partikelnummern angegeben werden, um Flugbahnen zu erhalten.

durchsucht. Partikel, die regularisiert wurden, erhalten für den Regularisierungszeitraum eine andere Identifikationsnummer. Sie werden berücksichtigt, indem die Suche ab dem Zeitpunkt der Regularisierung rekursiv nach der Identifikationsnummer des regularisierten Partikels fortgesetzt wird. Hierdurch werden auch mehrfach regularisierte Partikel erfasst.

Weitere Abfragemöglichkeiten

In der Visualisierung können unter Umständen Informationen von Interesse sein, die aus den beiden bisher besprochenen Abfragemöglichkeiten nicht hervorgehen. Dazu gehören

- Minimum- und Maximumwerte der einzelnen Attribute aller im Speicher befindlichen Datensätze,
- die Normen der Minimum- und Maximumwerte sowie
- die Anzahl der Partikel.

Bei der Speicherung der Daten werden diese Werte daher aktuell gehalten und können so jederzeit abgefragt werden.

5.3 Bereitstellung der Daten

Im Zwischenspeicher wird eine Menge von Datensätzen, die zu unterschiedlichen Zeitpunkten gehören, abgespeichert. In der Visualisierung wird jedoch meist nur auf die Datensätze, die zum gegenwärtig betrachteten Zeitpunkt gehören, zugegriffen. Daher wurde ein Modul (*DataReader*) entworfen, welches diese Datensätze innerhalb der Visualisierung aus dem Zwischenspeicher in eine für die Visualisierung günstige Datenstruktur kopiert. So entsteht eine Momentaufnahme von den Daten des Zwischenspeichers zu einem bestimmten Zeitpunkt (vgl. Abbildung 5.5 auf der nächsten Seite). Die Attribute aller Sterne zu einem Simulationszeitpunkt können über dieses Modul als eigenständige Datenfelder angesprochen werden.

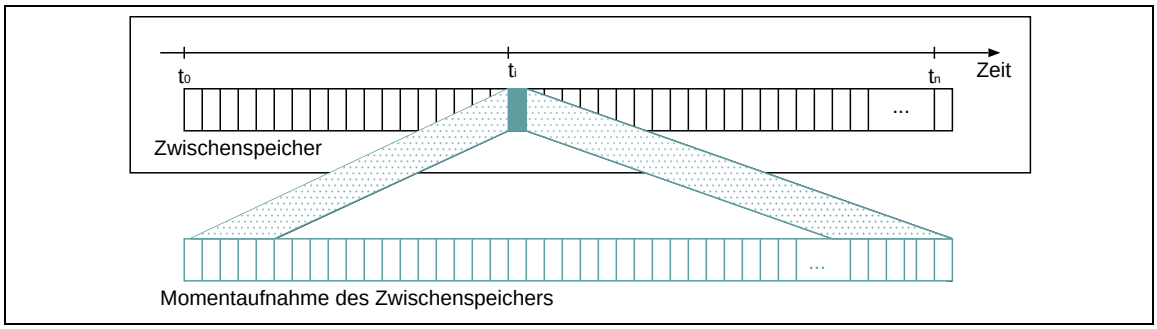


Abbildung 5.5: Das Modul *DataReader*: Dieses Modul erstellt einen Ausschnitt des Zwischenspeichers, in dem nur Datensätze, die zu dem betrachteten Simulationszeitpunkt gehören, betrachtet werden.

Eine weitere Funktion ermöglicht es, die Identifikationsnummern von Partikeln anhand von Beschränkungen bestimmter Attribute herauszusuchen.

5.4 Integration von VISIT

Da die Visualisierung bei der Kommunikation mit der Simulation die Rolle eines Servers übernimmt, müssen hier auch alle Initialisierungen ausgeführt werden, die nötig sind, um eine Verbindung mit der Simulation (Client) aufzubauen. Danach muss der Service auf dem Seap-Server registriert werden. Der Client kann dann dort die Informationen abfragen, die nötig sind, um eine Verbindung zum Server aufzubauen.
VISIT ermöglicht, wie in Kapitel 2 bereits erwähnt, eine bidirektionale Kommunikation zwischen der Simulation und der Visualisierung.

In der einen Richtung werden Daten von der Simulation zur Visualisierung geschickt (Online-Visualisierung). Die Kommunikation wird mit Hilfe von Sockets realisiert. VISIT verwendet hierfür zwei Sockets: einen, der der Kontrolle der Kommunikation dient, und einen, über den die Daten ausgetauscht werden.

Wie in Kapitel 2 beschrieben, bietet Qt die Möglichkeit, auf Socket-Events zu reagieren. Die Events werden in die Eventqueue eingereiht, wodurch eine nicht-blockierende Kommunikation gewährleistet wird. Daher ist die Verwendung von Threads nicht notwendig. In der anderen Richtung können Parameter für die laufende Simulation übertragen werden (Steering). Um die Simulation so wenig wie möglich zu stören, wird auch diese Kommunikationsrichtung von der Simulation angestoßen.

5.5 3-dimensionale Darstellung der Daten

Für die 3-dimensionale Darstellung werden die folgenden Eigenschaften der Sterne visualisiert.

Positionen der Partikel

Um die Positionen der Partikel im Raum darzustellen, müssen graphische Objekte an jeder dieser Positionen angezeigt werden.
Als graphische Objekte können entweder Kugeln oder Punkte gewählt werden. Bei der Darstellung durch Kugeln besteht die Möglichkeit, deren Radien anhand frei wählbarer Attribute zu skalieren. Die Kugeln werden durch Polyeder approximiert. Approximationsqualität und Radius der Kugeln können explizit angegeben werden.

Durch die Anzeige einer *Bounding Box*, die alle Sterne umgibt, und eines drei-dimensionalen Koordinatenkreuzes können die Koodinaten der Sterne eingeschätzt werden.

Geschwindigkeiten

Die Geschwindigkeiten werden durch Pfeile dargestellt, deren Ursprung in der Position des dazugehörigen Partikels liegt und deren Orientierung entlang des Geschwindigkeitsvektors verläuft. Die Geschwindigkeitspfeile werden immer anhand der Geschwindigkeitsvektoren skaliert. Es kann eine Minimalgeschwindigkeit angegeben werden, ab welcher die Pfeile erst angezeigt werden.

Darstellung verschiedener Attribute

Es gibt zwei verschiedene Arten der Darstellung von Attributen. Zum einen können, wie oben beschrieben, die Größen der Kugeln Aufschluss über bestimmte Eigenschaften der Partikel geben.
Zum anderen können die Objekte auch anhand von verschiedenen, vom Benutzer wählbaren Attributen eingefärbt werden. In beiden Fällen muss in der Darstellung herausgestellt werden, auf welche Eigenschaft sich die Größen bzw. Farben der Kugeln beziehen. Zum Beispiel kann die Geschwindigkeit auf die Farben der Kugeln abgebildet werden, während die Größe die Dichte widerspiegelt. Für die Interpretation der Darstellung werden Legenden benötigt. Die Einordnung der Farben erfolgt über Farbtafeln, die im Titel das dargestellte Attribut anzeigen. Die Skala der Farbtafel ist an das jeweilige Attribut angepasst.
Im Gegensatz zu den Farben lassen die Größen der Kugeln nur eine qualitative Interpretation zu, da durch die perspektivische Darstellung eine genaue Größenzuordnung nicht möglich ist. Daher ist eine Skala an dieser Stelle nicht sinnvoll. Der Zusammenhang zwischen Skalierung und Attribut wird lediglich durch eine Anzeige des Attributnamens angegeben.

Flugbahnen

Die Flugbahnen der Partikel werden durch einfache Polygonzüge dargestellt. Die Farbgebung dieser Polygonzüge kann wiederum durch die zu den Positionen gehörigen Attributwerte festgelegt werden. An den gegebenen Positionen werden die zu den Attributen gehörenden Farben angenommen. Auf den Abschnitten dazwischen wird ein Farbverlauf erzeugt.
Wie in Kapitel 4.4 bereits beschrieben, wird dem Benutzer die Möglichkeit gegeben, den Wertebereich der abzubildenden Attribute selbst festzulegen. Die Farben der Objekte werden anhand von Tabellen bestimmt. In diesen Tabellen muss ein Intervall angegeben werden, in dem die Attributwerte liegen. Der Benutzer kann als Intervallgrenzen den Wertebereich des Attributs über den gesamten Speicher oder den gerade betrachteten Simulationszeitpunkt wählen. Alternativ hierzu kann er auch explizit ein Intervall angeben.

Interaktion mit der Graphik

Die Visualisierung bietet dem Benutzer die Möglichkeit mit der Graphik zu interagieren. Zum Beispiel kann er sie drehen und hinein- bzw. herauszoomen. Dadurch wird dem Benutzer eine bessere räumliche Vorstellung der Posistionen der Sterne vermittelt.

5.6 2-dimensionale Darstellung der Daten

Bei der 2-dimensionalen Darstellung der Simulationsdaten wählt der Benutzer Orts- und Geschwindigkeitskomponenten aus, die gegeneinander abgebildet werden. Hierzu wird ein Koordinaten-

system erzeugt, dessen Achsen mit den Namen der gewählten Komponenten beschriftet werden. Die Daten werden von dem Modul *DataReader* abgefragt. Die Darstellung erfolgt als Punktwolke.

5.7 Die graphische Benutzeroberfläche

Die Interaktion zwischen dem Benutzer und der Visualisierung wird durch die graphische Oberfläche realisiert. Sie wird logisch und optisch in mehrere Komponenten gegliedert (vergl. Abbildung 5.6).

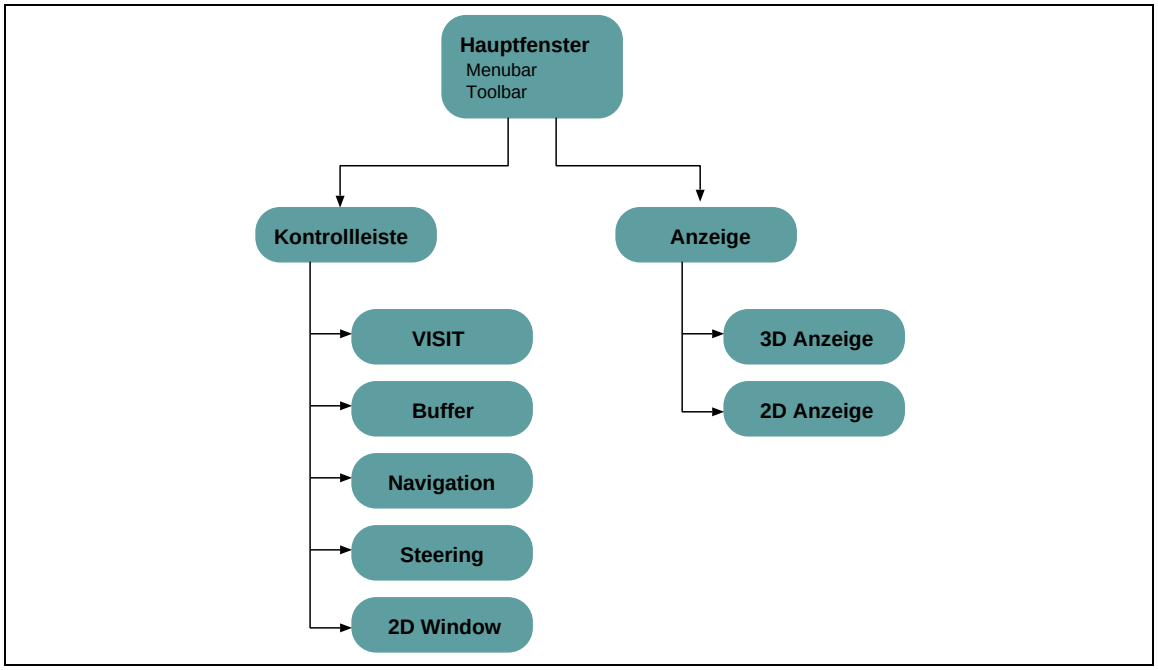


Abbildung 5.6: Logischer Aufbau der graphischen Benutzeroberfläche: Es wird ein Hauptfenster erzeugt, in welches die Kontrollleiste sowie die Anzeige der Daten integriert werden. Sowohl die Kontrollleiste als auch die Anzeige werden in weitere speziellere Komponenten gegliedert.

Es wird ein Hauptfenster erzeugt, welches mit einer Menüleiste und einer Toolbar ausgestattet ist. In dem Hauptfenster werden eine Kontrollleiste und ein Anzeigefenster integriert. Die Kontrollleiste kann nochmals unterteilt werden in Komponenten zur Steuerung und/oder Anzeige der einzelnen Module:

- VISIT
  Hier werden Eingabefelder für die Parameter, die für den Aufbau einer Verbindung notwendig sind, bereitgestellt. Außerdem werden Werkzeuge zum Verbindungsaufbau und -abbau sowie eine Statusanzeige angeboten.
- Buffer
  Diese Komponente gibt Aufschluss über den Zustand des Zwischenspeichers. Die Größe und der Füllstand können angezeigt werden.
- Navigation
  Hier werden Vorrichtungen zur Navigation innerhalb des Zwischenspeichers angeboten. Mit Hilfe dieser Vorrichtungen kann der Benutzer durch die im Zwischenspeicher enthaltenen Daten „fahren“ und Daten zu verschiedenen Zeitpunkten abfragen.
- Steering
  Der Benutzer kann in die hier gegebenen Eingabefelder Steering-Parameter eintragen, die dann bei der nächsten Steering-Nachricht an die Simulation geschickt werden.

- 2D Window
  Hier kann der Benutzer auswählen, welche Orts- bzw. Geschwindigkeitskomponenten in die 2-dimensionalen Darstellung eingehen.

Auch das Anzeigefenster gliedert sich nochmals in zwei Bereiche, in denen die 3- und 2-dimensionalen Graphiken angezeigt werden. Hierbei kommt der 3-dimensionalen Anzeige eine größere Gewichtung und somit auch mehr Raum zu.

In Abbildung 5.7 wird die Anordnung der Komponenten in der graphischen Oberfläche dargestellt.

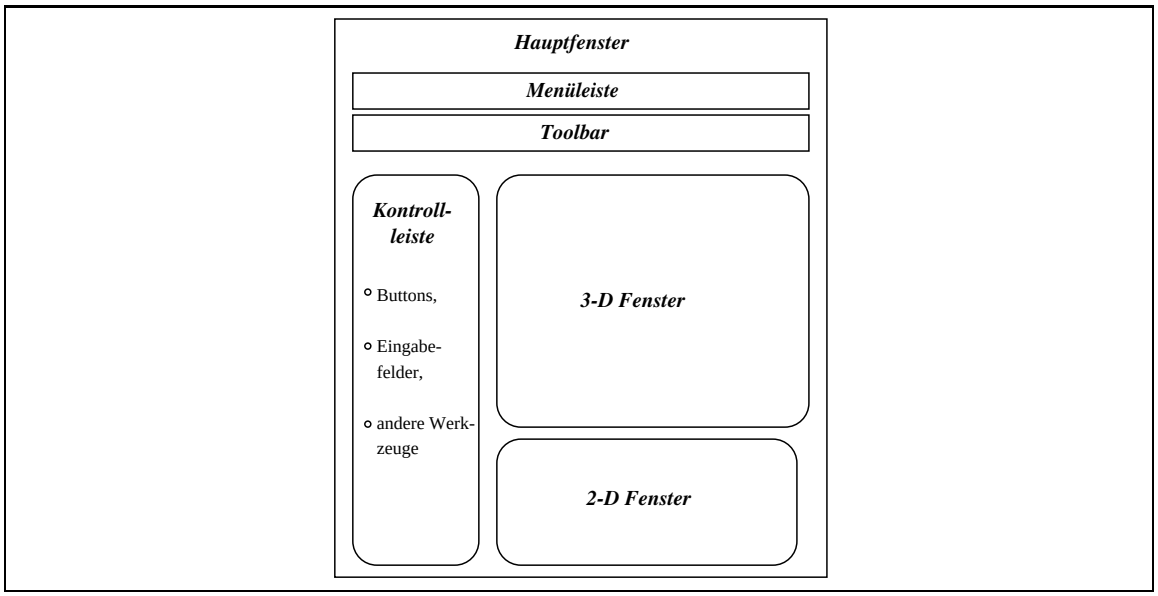


Abbildung 5.7: Entwurf der graphischen Oberfläche: Die Oberfläche wird unterteilt in Bereiche für die Anzeige der Daten und für die Benutzerinteraktion. Menüleiste, Werkzeugleiste sowie die Kontrollleiste werden zur Benutzerinteraktion verwendet. Zur Anzeige der Daten werden ein 3-dimensionales und ein 2-dimensionale Fenster erzeugt.

Am oberen Rand befinden sich eine Menüleiste und eine Werkzeugleiste, über die der Benutzer bestimmte Aktionen ausführen kann, wie zum Beispiel von der Darstellung durch Kugeln zur Darstellung durch Punkte wechseln. Darunter befindet sich links eine Kontrollleiste, in der die Eingabefelder, Schalter und Statusanzeigen zu finden sind. Rechts daneben befindet sich ein Bereich für die Darstellung der Daten. Dieser gliedert sich nochmals in ein großes Fenster für die 3-dimensionale und ein kleineres Fenster für die 2-dimensionale Darstellung.

5.8 Informationsaustausch zwischen den einzelnen Modulen

Die in den Kapiteln 5.2 bis 5.7 beschriebenen Module tauschen untereinander Informationen aus. Handelt es sich bei diesen Informationen um Ereignisse (z.B. „Im Zwischenspeicher wurden neue Daten abgelegt“), so werden diese mit Hilfe des Signal-Slot-Mechanismus von Qt (vgl. Kapitel 2) an die Module, die sich für diese Ereignisse interessieren, weitergeleitet. Dabei muss beachtet werden, dass die Signale und Slots die gleiche Schnittstelle besitzen (vgl. Abbildung5.8).

Neben den Signalen und Slots werden auch andere (herkömmliche) Funktionen für den Informationsaustausch verwendet.

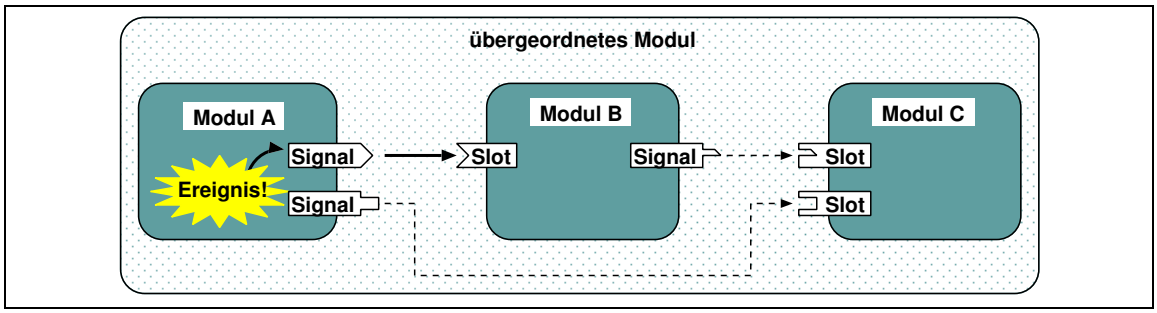


Abbildung 5.8: Beispiel für die Funktionsweise des Signal-Slot-Mechanismus: Modul A besitzt zwei Signale, Modul B sowohl ein Signal, als auch einen Slot und Modul C verfügt nur über zwei Slot. Die drei Module sind in ein übergeordnetes Modul integriert, welches dafür sorgt, dass die Verbindung zwischen den Signalen und den Slots hergestellt wird. In Modul A tritt ein Ereignis auf, woraufhin ein Signal emittiert wird. Dieses wird von Modul B empfangen.

Kapitel 6

Implementierung

In diesem Kapitel wird die Umsetzung des Konzeptes, d.h. die Implementierung der einzelnen Module, aus denen sich die Online-Visualisierung (*xnbody*) zusammensetzt, erläutert. Die Module und die Abhängigkeiten zwischen ihnen sind in der Abbildung 6.1 dargestellt.

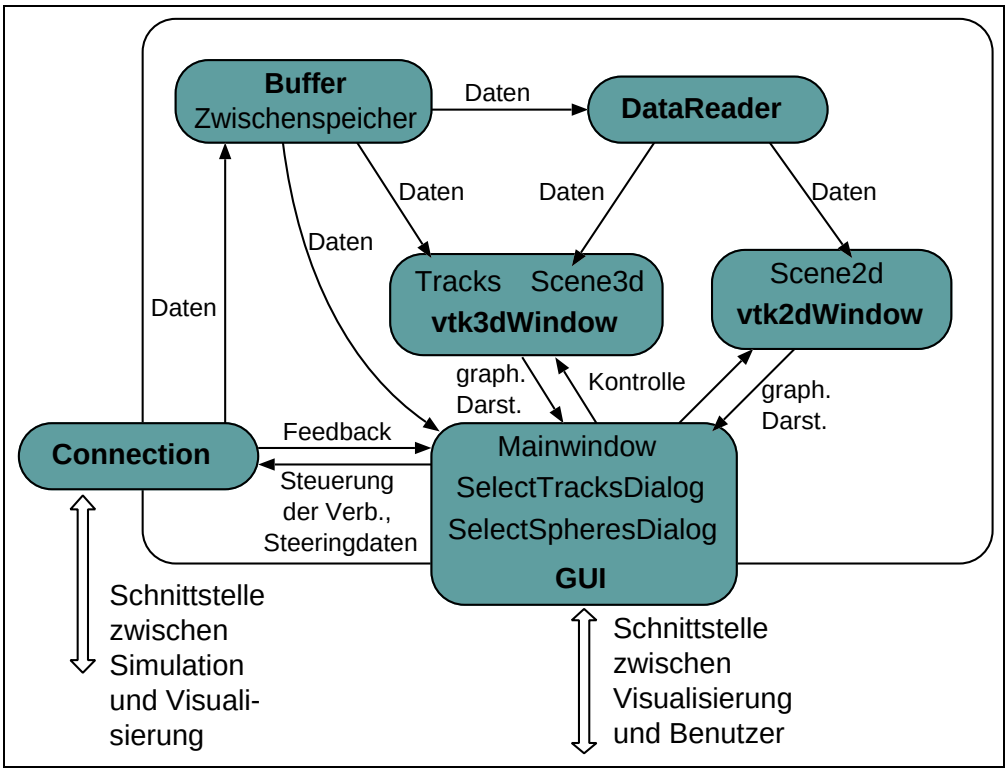


Abbildung 6.1: Abhängigkeiten zwischen den einzelnen Modulen von *xnbody*: *xnbody* verfügt über zwei Schnittstellen: eine zwischen Simulation und Visualisierung und eine für die Benutzerinteraktion.
Die Schnittstelle zur Simulation wird durch das Modul *Connection* realisiert. Über dieses Modul gelangen die Simulationsdaten in den Zwischenspeicher und Steering-Daten zur Simulation. Die Simulationsdaten werden von verschiedenen Modulen vom Zwischenspeicher abgefragt. Zum Beispiel das Modul *DataReader* liest Daten aus dem Zwischenspeicher, um sie für die 2- bzw. 3-dimensionalen Darstellung in einer günstigen Form bereitzustellen.
Das für die Benutzerinteraktion erstellte Modul *GUI* bietet die Möglichkeit der Kontrolle von fast allen anderen Modulen.

Zunächst wird der Zwischenspeicher vorgestellt, da er in *xnbody* eine zentrale Rolle spielt und fast alle anderen Module auf ihn zugreifen. Dann wird auf die Einbindung von VISIT eingegangen.

Über dieses Modul (*Connection*) gelangen die Simulationsdaten in den Zwischenspeicher. Bevor die Erzeugung der 3- bzw. 2-dimensionalen Darstellungen (*vtk3dWindow, vtk2dWindow*) der Sterne erläutert werden kann, wird ein Modul zur Bereitstellung der Daten für diese Darstellungen vorgestellt. Schließlich wird die Implementierung der graphischen Oberfläche erläutert, in welche die Darstellungen integriert werden. Die Verbindung zwischen Simulation und Visualisierung kann über dieses Modul kontrolliert werden.

6.1 Realisierung des Zwischenspeichers

Um auf die Implementierung des Zwischenspeichers näher einzugehen, werden zunächst die zentralen Datenstrukturen vorgestellt. Auf dieser Basis werden dann die Algorithmen erläutert. Da der Zwischenspeicher als eigenständiges Modul implementiert worden ist, muss er in die Visualisierung integriert werden. Hierzu wird zunächst die Schnittstelle des Zwischenspeichers (API[1]) und dann die Einbindung in die Visualisierung vorgestellt.

6.1.1 Datenstrukturen

Die Grundstruktur des Zwischenspeichers (*storedata*) besteht sowohl aus Komponenten, die der Verwaltung des Speicherplatzes und somit der Realisierung der FIFO-Struktur dienen, als auch aus Komponenten, die für die Verwaltung der Daten notwendig sind (vgl. Kapitel 5.2).
Für die Speicherplatzverwaltung werden die Kapazität und die Auslastung des Speichers protokolliert. Ein Zeiger auf ein Datenfeld ermöglicht später den Zugriff auf die Daten im Zwischenspeicher. Für die Verwaltung der Daten werden die Werte *ifirst* und *ntot* gespeichert, die für den Umgang mit KS-regularisierten Partikeln benötigt werden (vgl. Abbildung 6.2, Teil 1).

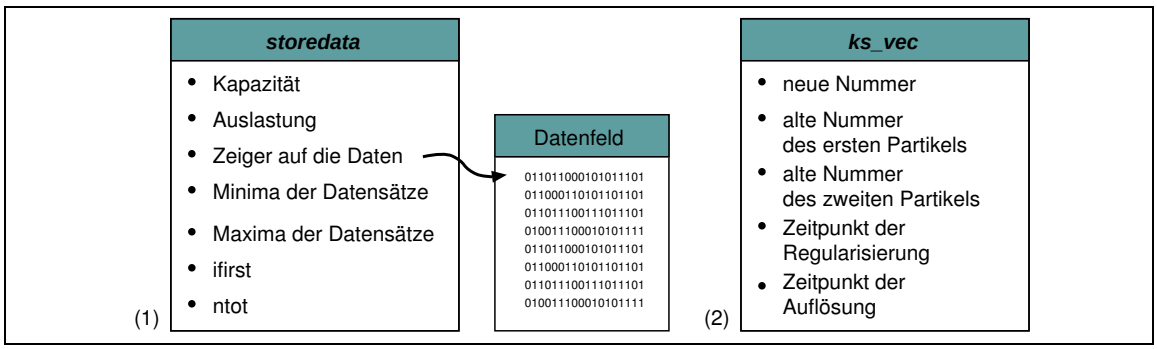


Abbildung 6.2: Datenstrukturen des Zwischenspeichers: In der Struktur *storedata* werden die Simulationsdaten, aber auch wichtige Eigenschaften des Zwischenspeichers und der Daten abgespeichert. Die Struktur *ks_vec* ist für den Umgang mit KS-Regularisierungen zuständig.

Die Bedeutung der beiden Variablen ergibt sich aus der internen Datenstruktur von NBODY6++ (vgl. Abbildung 6.3). In NBODY6++ werden alle zu einem Simulationszeitpunkt gehörenden Datensätze in einem Feld abgespeichert. Dabei werden bei einer KS-Regularisierung die beiden regularisierten Partikel in den vorderen Teil des Feldes kopiert. Der Wert *ifirst* enthält den Index des ersten nicht regularisierten Partikels. Das neu erzeugte Partikel wird an das Ende des Feldes kopiert. Die Variable *ntot* enthält die Gesamtanzahl aller Partikel, inklusive der KS-regularisierten Partikel. Bei der Übertragung der Daten zur Visualisierung werden die Datensätze *0* bis *ifirst-1* ausgelassen. Lediglich die Datensätze *ifirst* bis *ntot* werden übertragen.
Anhand der Werte *ifirst* und *ntot* kann zu jedem im Speicher vorliegenden Zeitpunkt die Anzahl

[1] Application Programming Interface

der Partikel bestimmt werden.

Die Minima und Maxima sowie die Normen der Attribute werden ebenfalls in der Struktur *storedata* gespeichert. Dies erleichtert später den Zugriff auf diese Daten.

Wie in Kapitel 5.2 beschrieben, wird für die Regularisierung eine weitere Datenstruktur (*ks_vec*) angelegt. In dieser Datenstruktur werden die Namen der zusammengeführten Partikel, der neue Name sowie die Zeitpunkte der Regularisierung und der Auflösung gespeichert (siehe Abbildung 6.2 auf der vorherigen Seite, Teil 2).

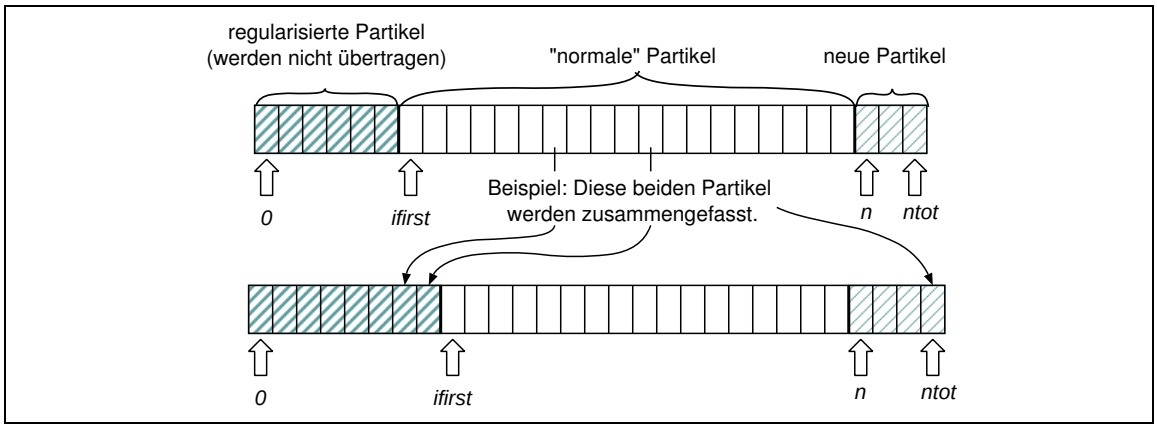


Abbildung 6.3: Datenstruktur auf der Simulationsseite: Auf der Simulationsseite werden die Datensätze, die zu einem Simulationszeitpunkt gehören, in einem Feld abgespeichert. Regularisierte Partikel werden in den vorderen Teil des Feldes verschoben, sie werden bei der Übertragung zur Visualisierung ausgelassen. Die aus einer Regularisierung hervorgehenden Partikel werden am Ende des Feldes gespeichert. Die Werte *ifirst* und *ntot* sind Indizes auf das erste nicht regularisierte Partikel und das letzte Partikel insgesamt.

6.1.2 Algorithmen

In diesem Unterkapitel wird auf die Implementierung der in Abbildung 5.3 auf Seite 21 dargestellten Funktionen zur Realisierung des Zwischenspeichers (*init*, *push*, *get* und *close*) eingegangen. Die Algorithmen basieren auf den im vorigen Kapitel erläuterten Datenstrukturen.

Initialisierung und Allokation des Zwischenspeichers (*init*)

Bei der Initialisierung werden die Hauptdatenstrukturen (Abbildung 6.2 auf der vorherigen Seite) angelegt. Mit Hilfe der Bibliothek VTools, die auch ein Teil der VISIT-Bibliothek ist, wird der Adresse der Struktur ein Integer-Deskriptor zugeordnet. Dieser Deskriptor wird an alle Funktionen, die auf den Zwischenspeicher zugreifen, übergeben. In den Funktionen wird dann wiederum mittels Funktionen aus VTools die Adresse der Struktur ermittelt. Durch diese Technik kann auf die Übergabe von Zeigern auf den Zwischenspeicher verzichtet werden. Dies spielt im Hinblick auf eine Portierung auf andere Programmiersprachen (z.B. FORTRAN) eine Rolle. Außerdem erleichtert sie die Verwendung von mehreren Zwischenspeichern in einem Programm.

Als nächstes wird für den Datenbereich Speicherplatz angelegt und Kapazität und Auslastung werden initialisiert.

Abspeichern von Datensätzen (*push*)

Bei der Speicherung der Datensätze müssen folgende im Konzept beschriebenen Situationen berücksichtigt werden:

1. Im Speicher ist ausreichend Platz vorhanden für die zu speichernden Datensätze.
2. Im Speicher sind bereits so viele Daten gespeichert, dass der Platz nicht mehr ausreicht.
3. Der Speicher ist grundsätzlich zu klein für die Menge der Datensätze.

Im ersten Fall können die Daten direkt hinter eventuell vorhandenen Datensätzen gespeichert werden. Bei der Speicherung werden die Minima, Maxima und Normen der Attribute aktualisiert, damit diese Werte bei einer Abfrage zur Verfügung stehen und nicht erst berechnet werden müssen. Auch die Einträge in der Struktur für die KS-Regularisierung müssen aktuell gehalten werden. Daher werden Datensätze des Typs *KSREG* und *KSTERM* betrachtet und registriert.
Der nächste Fall erfordert eine Ersetzung von alten Datensätzen. Wenn der Speicher nicht vollständig belegt ist, muss zunächst berechnet werden, wieviele Datensätze ersetzt werden müssen. Hierfür wird die Anzahl der „freien Plätze“ aus der Kapazität und der Auslastung ermittelt. Die Differenz zwischen dieser und der Anzahl der zu speichernden Datensätze muss aus dem Speicher entfernt werden. Praktisch geschieht dies durch eine Überschreibung der alten Daten mit den Datensätzen, die noch im Speicher verbleiben (siehe Abbildung 6.4).
Bevor die Daten gelöscht werden, wird ein Array erstellt mit den Datensätzen, die zu dem frühesten Zeitpunkt gehören, zu dem Datensätze im Zwischenspeicher vorliegen. Hierdurch wird garantiert, dass zu jedem Partikel mindestens ein Datensatz vorliegt, mit Hilfe dessen eine Vorausberechnung erfolgen kann.

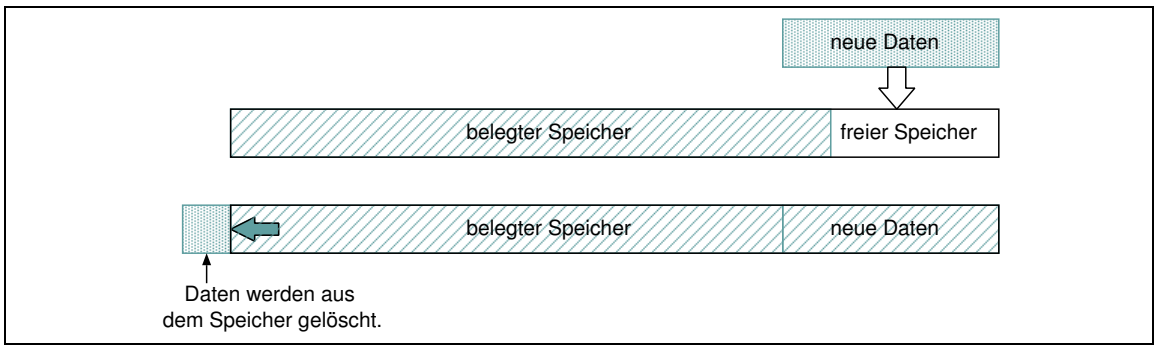


Abbildung 6.4: Ersetzungsstrategie für die Daten des Zwischenspeichers: Bevor neue Daten in dem Zwischenspeicher abgelegt werden können, muss eventuell Platz geschaffen werden. Hierzu werden die Datensätze, die aus dem Speicher entfernt werden sollen, mit denen, die noch im Speicher bleiben sollen, überschrieben. Anschließend werden in dem frei gewordenen Speicher die neuen Datensätze abgelegt.

Bevor Daten überschrieben werden können, wird ermittelt, ob darin Regularisierungen vorkommen. Falls unter den zu überschreibenden Daten ein Datensatz mit dem Typ *KSTERM* ist, durch den eine Regularisierung abgeschlossen wird, werden die dazugehörigen Daten in der Struktur für die KS-Regularisierungen ebenfalls entfernt. Anschließend kann verfahren werden wie im ersten Fall.
Die vollständige Belegung des Zwischenspeichers stellt einen Sonderfall des zweiten Falles dar und wird analog behandelt.
Ist der dritte Fall gegeben, können nicht alle Datensätze gespeichert werden. Die Kapazität des Zwischenspeichers bestimmt in diesem Fall, wieviele Datensätze am Anfang des abzuspeichernden Blockes ignoriert werden müssen.
In jedem Fall müssen die Auslastung, *ntot* und *ifirst* aktualisiert werden. Die Werte *ntot* und *ifirst* werden durch Datensätze mit den Typen *KSREG, KSTERM* und *STAT* übermittelt.

Abfragen von Daten (*get*)

Die Abfrage der Datensätze kann auf zwei Arten erfolgen. Es können Datensätze zu

- allen Sternen zu einem Zeitpunkt oder
- mehreren Sternen über einen im Speicher befindlichen Zeitraum (Flugbahnen)

abgefragt werden.

Abfrage nach einem Zeitpunkt

Bei der Abfrage nach einem Zeitpunkt muss zunächst die Stelle im Speicher gefunden werden, an der die Datensätze mit dem entsprechenden Zeitwert liegen. Dafür wird der Speicher von hinten durchsucht und die Position bestimmt, an der der Zeitwert zum letzten Mal echt größer ist als der zu suchende Zeitpunkt.
Um die Anzahl der zu suchenden Partikel zu ermitteln, werden die Werte *ifirst* und *ntot* aus der Hauptdatenstruktur benötigt. Diese sind jedoch nur für den zuletzt gespeicherten Zeitpunkt gültig. Bei dem Durchlaufen des Speichers wird daher bei Antreffen eines *KSTERM*-Datensatzes *ntot* um 1 und *ifirst* um 2 erhöht. Analog wird bei einem Datensatz vom Typ *KSREG ntot* um 1 und *ifirst* um 2 erniedrigt. Ist die Stelle im Speicher, an der die Datensätze mit dem gesuchten Zeitpunkt vorliegen, erreicht, so ergibt sich die Anzahl der Partikel zu diesem Zeitpunkt aus $ntot - ifirst + 1$.

Als nächstes wird der Speicher von der gefundenen Stelle aus rückwärts durchlaufen. Dabei werden die Datensätze der Typen *REALPART* und *KSPART* betrachtet. Ist der dazugehörige Zeitwert kleiner als der gesuchte Zeitpunkt, so muss eine Vorausberechnung der Ortskoodinaten und Geschwindigkeiten durchgeführt werden.

Damit nicht mehrere Datensätze zu einem Partikel abgespeichert werden, wird durch eine eindeutige Zuordnung die Position im Ausgabefeld bestimmt. Jetzt muss abgefragt werden, ob im Ausgabefeld an der für das jeweilige Partikel bestimmten Position bereits ein Datensatz gespeichert wurde. Um nicht den gesamten Speicher durchsuchen zu müssen, wird die Anzahl der zu suchenden Partikel bei jedem Eintrag in das Ausgabefeld erniedrigt und die Suche gestoppt, wenn alle Partikel gefunden wurden (vgl. das Nassi-Shneiderman-Diagramm in Abbildung A.1).
Die Abfrage nach einem Zeitpunkt ist also von linearer Komplexität, da sie im Wesentlichen aus zwei Schleifen besteht, die nacheinander ausgeführt werden.

Abfrage von Flugbahnen

Bei der Abfrage nach einem einzelnen Stern in einem Zeitraum t_1 bis t_2 ist, wie im Konzept bereits erläutert, eine rekursive Funktion (*nbodybuf_getSingleTrack*) erforderlich, da regularisierte Partikel unter einer anderen Identifikationsnummer zu finden sind (vgl. Abbildung 6.5 auf der nächsten Seite).

Zu Beginn der Funktion muss mit Hilfe der Struktur *ks_vec* überprüft werden, ob das gesuchte Partikel regularisiert wurde. Falls das der Fall ist, ist der Zeitraum der Regularisierung interessant:

Falls die Regularisierung vor und die Auflösung nach dem Zeitpunkt t_1 eintritt, muss die Abfrage zunächst mit der Identifikationsnummer des KS-regularisierten Partikels durchgeführt werden.

Tritt die Regularisierung innerhalb des Zeitraumes t_1 bis t_2 auf, so werden bis zum Zeitpunkt der Regularisierung, der durch einen Datensatz des Typs KSREG gekennzeichnet wird, alle Datensätze des Partikels gespeichert. Von dort aus wird die Suche mit der Identifikationsnummer des KS-regularisierten Partikels rekursiv gestartet.

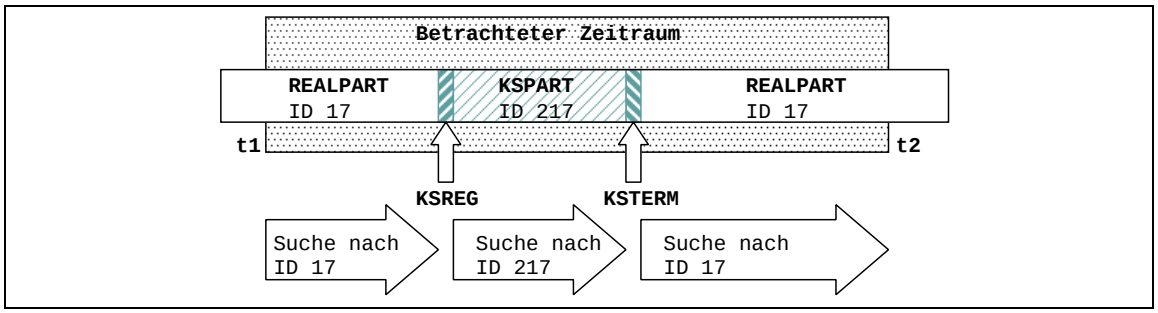


Abbildung 6.5: Ablauf einer Suche nach einem Stern über einen Zeitraum: Zunächst wird *nbodybuf_getSingleTrack* für das Partikel mit der Nummer 17 aufgerufen. Bei der Suche nach diesem Partikel wird eine Regularisierung festgestellt und *nbodybuf_getSingleTrack* wird erneut, diesmal mit der neuen Nummer (217) des regularisierten Partikels, aufgerufen. Die Funktion endet, sobald die Regularisierung aufgelöst wird. An dieser Stelle wird die Suche nach dem Partikel mit der Nummer 17 fortgesetzt. In der Abbildung werden aus Übersichtsgründen nur die gesuchten Datensätze dargestellt. Diese stehen im realen Zwischenspeicher zwischen anderen Datensätzen, die bei der Suche ignoriert werden.

Die Rückkehr in die aufrufende Funktion erfolgt, sobald der Zeitpunkt t_2 erreicht ist oder ein Datensatz mit dem Typ *KSTERM* mit der entsprechenden Identifikationsnummer gefunden wird. Im letzteren Fall wird die Suche dann fortgesetzt. Als Ergebnis der Suche wird ein Ausgabefeld mit allen gefundenen Datensätzen zurückgeliefert, in dem die Suchergebnisse in chronologischer Reihenfolge gespeichert sind (vgl. Abbildung A.2 auf Seite 60).
Auch die Abfrage von Flugbahnen erfolgt mit linearer Komplexität, da hier der Zwischenspeicher linear durchsucht wird.

Die Flugbahnen mehrerer Sterne werden nach dem Prinzip „Teile und herrsche!“ ermittelt. Dazu wird der oben beschriebene Algorithmus in eine übergeordnete Funktion (*nbodybuf_gettrack*) eingegliedert, die ihn nacheinander für jeden Stern ausführt. Der zusätzliche Aufwand, der dadurch entsteht, ist nicht signifikant, da die Anzahl der Flugbahnen aus Gründen der Übersicht und Performanz der Graphiksoftware beschränkt ist.

Schließen des Zwischenspeichers (*close*)

Der Zwischenspeicher wird geschlossen, indem der Speicherplatz aller verwendeten Strukturen freigegeben wird.

6.1.3 Die Schnittstelle des Zwischenspeichers (API)

In diesem Unterkapitel wird das API des Zwischenspeichers vorgestellt. Es werden die Funktionen erläutert, die für die Verwendung des Zwischenspeichers benötigt werden. Alle Funktionen, mit Ausnahme von *nbodybuf_init*, erhalten als Eingabeparameter den in Kapitel 6.1.2 beschriebenen Integer-Deskriptor.

int *nbodybuf_init*(long int size)
Der Funktion wird die gewünschte Größe des Zwischenspeichers übergeben. Sie legt den Speicherplatz an und gibt einen Integer-Deskriptor zurück.

int *nbodybuf_reinit*(int id, long int size)
Diese Funktion legt den Zwischenspeicher mit der übergebenen Größe neu an.

int *nbodybuf_close*(int id)
Die Funktion gibt den Speicherplatz wieder frei.

int *nbodybuf_push*(int id, float *data, int n1, int n2)
Um Daten in dem Speicher abzulegen, werden dieser Funktion ein Zeiger auf die Daten und die Anzahl der Datensätze und Attribute übergeben. Wenn die Daten erfolgreich im Zwischenspeicher abgelegt werden konnten, liefert die Funktion den Wert *1* zurück, andernfalls den Wert *0.*

int *nbodybuf_getdata*(int id,
float qtime,
int* qdo, float *min, float *max,
int n1, int n2,
float *data, int *npart, float *time, int *rc)

Zur Abfrage von Datensätzen zu einem Simulationszeitpunkt wird der Funktion *nbodybuf_getdata* als Eingabeparameter die gewünschte Simulationszeit übergeben. Außerdem können zu allen Attributen Minima und Maxima angegeben werden, durch die nur bestimmte Datensätze herausgefiltert und zurückgeliefert werden. Als Ausgabeparameter wird ein Feld für die Daten benötigt. Dieses muss in der aufrufenden Funktion angelegt werden. Die Größe dieses Feldes muss ebenfalls übergeben werden. Neben dem Ausgabedatenfeld wird der gefundene Zeitpunkt und die Anzahl der im Ausgabedatenfeld gespeicherten Daten zurückgegeben.

int *nbodybuf_gettrack*(int id,
float *particle_ids,
int n_tracks,
float t1, float t2,
float *tracklist, int size_of_tracklist,
float **trackpointer,
int *n_datasets)

Die Funktion *nbodybuf_gettrack* liefert die Flugbahnen der angegebenen Partikel. Hierzu werden eine Liste von Partikelnummern und der Anfangs- und Endzeitpunkt des gesuchten Zeitraumes benötigt.
Das Ergebnis dieser Funktion ist eine Liste von Flugbahnen, deren Datensätze hintereinander in einem Feld gespeichert sind. Zur Trennung der einzelnen Flugbahnen wird ein Vektor mit Zeigern auf den jeweils ersten Datensatz einer Flugbahn und ein Vektor mit der Anzahl der Datensätze der jeweiligen Flugbahn zurückgegeben (vgl. Abbildung 6.6 auf der nächsten Seite). Als Ausgabeparameter werden daher benötigt:

- ein Feld für die Datensätze,
- die Größe dieses Feldes,
- ein Feld von Zeigern,
- ein Feld für die Anzahl der Datensätze jeder Flugbahn.

Diese Felder müssen in der aufrufenden Funktion angelegt werden.

long int *nbodybuf_getbufsize*(int id)
Diese Funktion liefert die Größe des Zwischenspeichers. Außer dem Integer-Deskriptor hat diese Funktion keine Parameter.

int *nbodybuf_isEmpty*(int id)
Mit der Funktion *nbodybuf_isEmpty* kann abgefragt werden, ob Daten im Zwischenspeicher vorhanden sind.

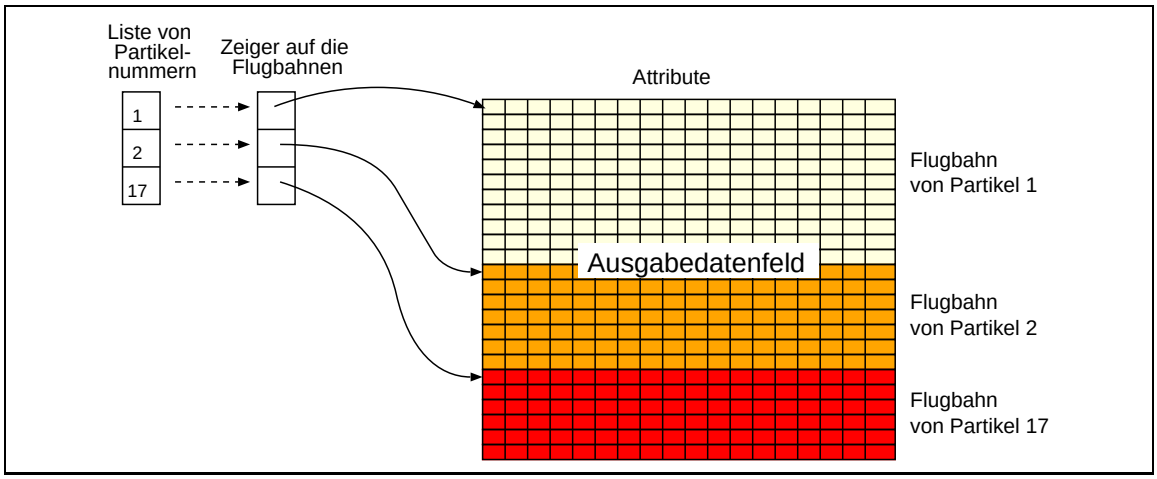


Abbildung 6.6: Rückgabeparameter der Funktion *nbodybuf_gettrack*: Der Funktion müssen verschiedene Felder zur Speicherung der Ergebnisse übergeben werden. Zum einen muss ein Feld für die Datensätze der Flugbahnen übergeben werden. Zur Interpretation dieser Daten wird noch ein weiteres Feld mit Zeigern auf die Anfänge der Flugbahnen übergeben. Die Zeiger werden den Partikelnummern durch ihre Position im Feld zugeordnet.

float *nbodybuf_getpercfull*(int id)
nbodybuf_getpercfull liefert die Auslastung des Zwischenspeichers in Prozent.

int *nbodybuf_getnPart*(int id)
Mit Hilfe dieser Funktion kann die Anzahl der Sterne, die simuliert werden, abgefragt werden.

int *nbodybuf_getminmax*(int id, float *min, float *max, int len)
Über die Funktion *nbodybuf_getminmax* können die Minimum- und Maximumwerte aller Attribute über alle Datensätze im Zwischenspeicher abgefragt werden. Der Funktion müssen Felder übergeben werden, in denen das Resultat der Funktion gespeichert werden kann.

int *nbodybuf_getminmaxnorm*(int id, float *minnorm, float *maxnorm, int len)
Anders als bei der Funktion *nbodybuf_getminmax* werden bei dieser Funktion die Minima und Maxima der Normen aller Attribute zurückgegeben. Dabei werden Attribute, die logisch zu einer Einheit gehören, wie zum Beispiel die Geschwindigkeitskomponenten, zu einem Vektor zusammengefasst. In jedem Fall wird die euklidische Norm gebildet.
Der Funktion muss also ebenfalls ein Feld für die Ausgabe übergeben werden.

6.1.4 Integration in die Visualisierung

Zur Integration des Zwischenspeichers in die Visualisierung wird eine Klasse *Buffer* implementiert, die auf die oben beschriebenen Funktionen des Zwischenspeichers zurückgreift.
Die Visualisierung soll gegebenenfalls aktualisiert werden, sobald neue Daten im Zwischenspeicher zur Verfügung stehen. Hierzu wird der Signal-Slot-Mechanismus von Qt (vgl. Kapitel 2) benutzt. In der Klasse *Buffer* wird immer dann, wenn neue Daten gespeichert wurden, ein Signal ausgesendet. Mit diesem Signal verbinden die Klassen, die auf die Ankunft neuer Daten reagieren müssen, bestimmte Funktionen.
Da mehrere Module auf den Zwischenspeicher zugreifen, wird die Klasse *Buffer* global instanziiert.

6.2 Einbindung von VISIT

Zur Einbindung von VISIT wird eine Klasse erzeugt, welche die Datenübertragung zwischen Simulation und Visualisierung steuert.

6.2.1 Die Klasse *Connection*

In der Klasse *Connection* wird ein VISIT-Server implementiert. Hierzu werden die in der Abbildung 6.7 dargestellten Funktionen verwendet. Die Klasse beinhaltet Variablen (*_snC* und *_snL*), welche die nicht blockierende Kommunikation mit der Simulation mit Hilfe von Events realisieren. Außerdem werden Funktionen (Slots) erzeugt, über die der Server gesteuert werden kann.

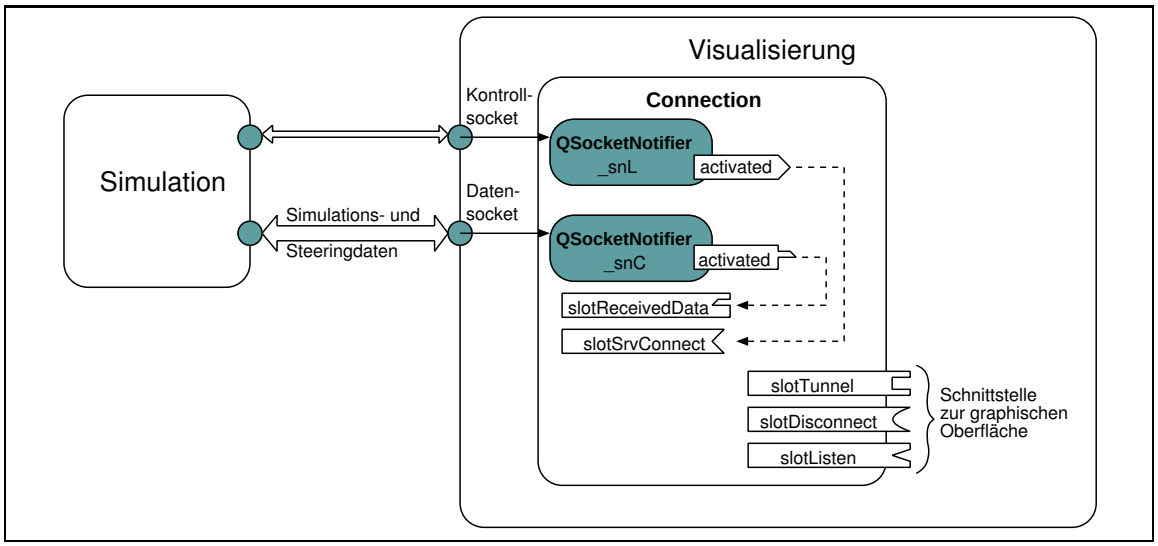


Abbildung 6.7: Einbindung von Visit durch die Klasse Connection: In der Klasse *Connection* werden die beiden *QSocketNotifier* *_snL* und *_snC* erzeugt und mit dem Kontroll- bzw. Datensocket der VISIT-Verbindung verbunden. *_snL* und *_snC* erzeugen jeweils ein Signal *activated*, durch welches der ihm jeweils zugeordnete Slot ausgeführt wird. Weitere Funktionen, die die Steuerung des VISIT-Servers ermöglichen, werden als Slots deklariert und dienen so als Schnittstelle zur graphischen Oberfläche.
In der Abbildung werden die anderen Komponenten der Visualisierung nicht aufgeführt.

Der Konstruktor

Im Konstruktor der Klasse *Connection* werden Initialisierungen vorgenommen. Zum Beispiel wird hier vereinbart, dass die Verbindung zunächst ohne *ssh-Tunnel* aufgebaut wird, falls der Benutzer diesbezüglich keine Eingaben macht.

Die Funktion *slotTunnel*

Besteht zwischen der Simulation und der Visualisierung eine Firewall, müssen die Daten über einen Tunnel ausgetauscht werden. In der Funktion slotTunnel wird ein ssh-Tunnel erzeugt, indem die VISIT-Funktion *vtt_open* aufgerufen wird. Die wichtigsten Parameter dieser Funktion wie der Hostname, die Userid und der Pfad zu dem Proxyprogramm, beziehen sich auf den Proxyserver, der die Daten entgegen nimmt. Dieser befindet sich zusammen mit der Simulation auf der einen Seite der Firewall. Zwischen dem Proxyserver und der Simulation wird nun weiterhin über Sockets kommuniziert.

Die Funktion *slotListen*

Die Funktion *slotListen* startet den VISIT-Server. Sie gibt die Informationen, die ein Client benötigt, um eine Verbindung mit dem Server aufzubauen, mit Hilfe der VISIT-Funktion *visit_srv_init_seap* auf dem *Seap-Server* bekannt und öffnet einen Listening-Port. Hierzu benötigt sie die vom Benutzer einzugebenden Parameter *service*, *passwd* und *interface*, wobei *interface* der Hostname ist. Die Funktion *visit_srv_init_seap* liefert einen *VISIT-Server-Descriptor* zurück, der in der Klasse *Connection* abgespeichert wird. Dieser Deskriptor wird für alle anderen VISIT-Funktionen benötigt. Um auf die Anfragen des Clients reagieren zu können, wird eine Instanz von *QSocketNotifier* (*_snL*) erzeugt, die mit dem Deskriptor des Kontrollsockets initialisiert wird. Wenn ein Client versucht, eine Verbindung aufzubauen, ist eine Aktivität auf dem Kontrollsocket zu verzeichnen und *_snL* sendet ein Signal (*activated*) aus. Dieses Signal wird mit dem Slot *slotSrvConnect* verbunden, der darauf entsprechend reagiert.

Die Funktion *slotSrvConnect*

Wird eine Aktivität auf dem Kontrollsocket festgestellt, so wird die Funktion *slotSrvConnect* ausgeführt. Diese Funktion baut die Verbindung zu dem Client auf, indem sie die VISIT-Funktion *visit_srv_connect* aufruft. Anschließend wird *_snL* deaktiviert und ein neuer *QSocketNotifier* (*_snC*) wird erzeugt. Dieser wird mit dem Socket, über den die Daten ausgetauscht werden, verbunden. *_snC* sendet bei jeder Nachricht, die über diesen Socket kommt, das Signal *activated* aus. Das Signal wird mit der Funktion *slotReceivedData* verknüpft.

Die Funktion *slotReceivedData*

Diese Funktion nimmt die Daten an, die von der Simulation an die Visualisierung geschickt werden. Die Datenpakete bestehen aus einem „Umschlag“ und den Simulationsdaten. Es gibt mehrere Typen von Datenpaketen. In dem „Umschlag“ werden unter anderem der Typ und die Dimensionen des Datenpaketes angegeben. Dieser kann mit Hilfe der VISIT-Funktion *visit_srv_get_id* in Erfahrung gebracht werden. Der Inhalt der Pakete wird über die Funktion *visit_srv_read_data* empfangen.
Für die Visualisierung sind vor allem die Typen „Partstep“ und „Steering“ relevant. Datenpakete mit diesen Typen werden wie folgt behandelt:

- **Partstep**
  Ein Datenpaket dieses Typs enthält Simulationsdaten. Die Daten werden direkt im Zwischenspeicher abgelegt.
- **Steering**
  Bei diesen Datenpaketen handelt es sich um leere „Umschläge“, die mit Steering-Daten gefüllt werden können. Tritt ein solches Datenpaket auf, werden die Steering-Parameter, die der Benutzer mit Hilfe der graphischen Oberfläche angeben kann, in diesem Umschlag zurück an die Simulation geschickt.

In jedem Fall wird nach Beendigung der Übertragung eine Empfangsbestätigung (*ack2*) an die Simulation gesendet. Hierdurch wird signalisiert, dass die Visualisierung bereit ist, neue Daten zu empfangen.

Die Funktion *slotDisconnect*

Die Funktion *slotDisconnect* schaltet den Server über die Funktion *visit_srv_shutdown* ab. Die beiden *QSocketNotifier* *_snC* und *_snL* werden deaktiviert.

6.3 3-dimensionale Darstellung der Daten

Bei der 3-dimensionalen Darstellung der Daten werden die Darstellung aller Sterne zu einem Zeitpunkt (in der Klasse *scene3d*) und die Darstellung der Flugbahnen einzelner Sterne (in der Klasse *Tracks*) getrennt behandelt. Beide Darstellungen werden jedoch von der Klasse *vtk3dWindow* gesteuert.

6.3.1 Die Klasse *scene3d*

Die Klasse *scene3d* erzeugt eine 3-dimensionale Graphik, die mit Hilfe einiger Funktionen modifiziert werden kann. Im Folgenden werden diese Funktionen vorgestellt.

Der Konstruktor

Im Konstruktor der Klasse *scene3d* werden im Wesentlichen die folgenden Objekte erzeugt, die dann in weiteren Funktionen der Klasse zu einer Visualisierungs-Pipeline miteinander verbunden werden:

Objekt	Funktion
vtkUnstructuredGrid	Die Klasse *vtkUnstructuredGrid* kann sehr unterschiedliche Datensätze repräsentieren. Um die Art des Datensatzes festzulegen, werden die Struktur, das heißt die Topologie und Geometrie der Daten, und Attributwerte angegeben.
vtkPolyVertex	Hiermit wird die Topologie der Daten festgelegt. Es handelt sich um eine Liste von Punkten im Raum, die nicht in Beziehung zueinander stehen.
vtkPoints	Mit Hilfe dieser Klasse wird die Geometrie der Datensätze angegeben, das heißt, die in der Topologie angegebene Struktur wird mit konkreten Werten instanziiert. Des Weiteren werden den einzelnen Punkten Attribute zugeordnet.
vtkSphereSource, *vtkPointSource* bzw. *vtkArrowSource*	Diese Objekte (Kugeln, Punkte und Pfeile) sollen bei der Visualisierung die Positionen und Geschwindigkeiten repräsentieren.
vtkGlyph3D	Ein *Glyph* kopiert ein graphisches Objekt an jeden Punkt, der in seinem Eingabedatensatz angegeben wurde.
vtkOutlineFilter	Dieser Filter berechnet die Eckpunkte des kleinsten Quaders, der sämtliche Objekte, in diesem Fall die Sterne, umschließt. Er wird benötigt, um die sogenannte *Bounding-Box* zu erzeugen.
vtkCubeAxesActor2D	In dieser Klasse werden Koordinatenachsen erzeugt. Die Achsen werden auf den Kanten der Bounding-Box gezeichnet.
vtkPolyDataMapper	Durch den Mapper werden die Daten in graphische Primitiven umgewandelt.

Es werden Initialisierungen vorgenommen, zum Beispiel wird festgelegt, dass zur Darstellung der Positionen zunächst Punkte angezeigt werden.

Die Funktion *SetNewScene*

In der Klassenfunktion *SetNewScene()* wird die Visualisierungs-Pipeline aufgebaut, indem die im Konstruktor erzeugten Objekte miteinander verbunden werden (vgl. Abbildung 6.8).

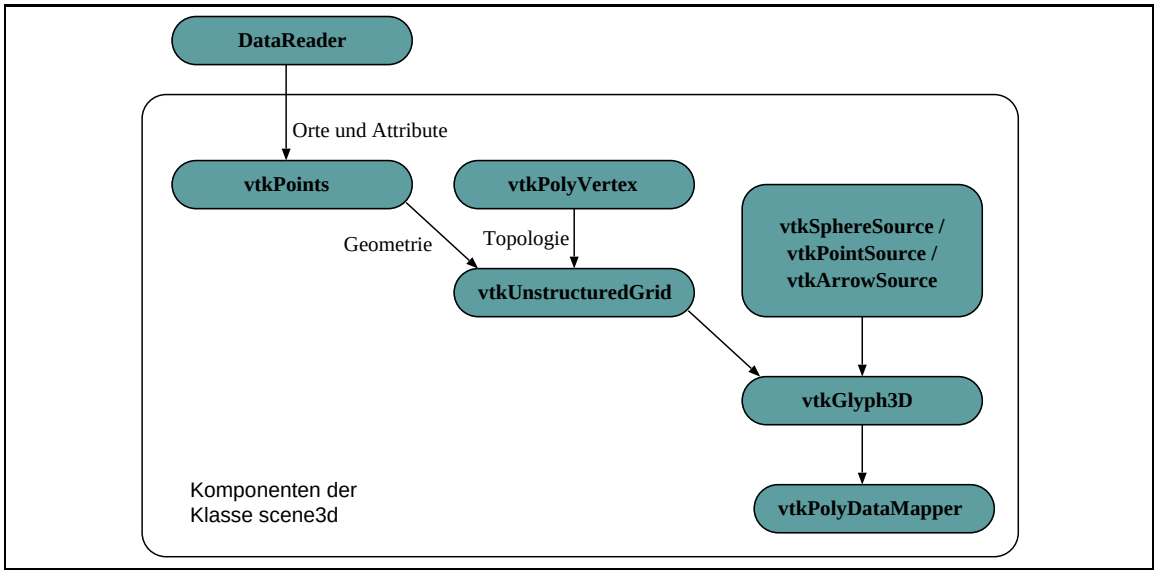


Abbildung 6.8: Visualisierungs-Pipeline der Klasse scene3d

Die Koordinaten und Attribute der Sterne liefert die Klasse *DataReader*. Diese werden dann in der Klasse *vtkPoints* abgespeichert und an die Klasse *vtkUnstructuredGrid* weitergegeben. Als Attribute werden den einzelnen Sternen sowohl Vektoren als auch Skalare zugeordnet.
Nun muss eine Topologie festgelegt werden, die beschreibt, in welcher Beziehung die angegebenen Punkte zueinander stehen. Hierzu wird eine Instanz der Klasse *vtkPolyVertex* erzeugt. Die Topologie wird dann der Klasse *vtkUnstructuredGrid* zugewiesen. Die Instanz von *vtkUnstructuredGrid* enthält jetzt also einen Satz von Punkten mit Attributen. Hierdurch werden in der Klasse *vtkGlyph3D* die Orte festgelegt, an welche die Kugeln, Punkte oder Pfeile kopiert werden.
Die Klasse *vtkGlyph3D* führt auch die Skalierung der Objekte nach einem Attribut sowie die Abbildung der Attributwerte auf Farben durch eine Farbtabelle aus.

Zur Darstellung der Positionen und Geschwindigkeiten der Sterne wird jeweils eine eigene Visualisierungs-Pipeline erzeugt. Die Positionen können sowohl als Kugeln als auch als Punkte dargestellt werden. Dazu werden beide Geometrien erzeugt, da sie nicht viel Speicherplatz benötigen. In die Visualisierungs-Pipeline eingefügt - und damit angezeigt - wird allerdings immer nur die gerade erwünschte Geometrie.
Die Geschwindigkeiten werden mit Hilfe der Klasse *vtkArrowSource* als Pfeile dargestellt, deren Ursprung in den Positionen der Partikel liegen und die mit dem Betrag der Geschwindigkeiten skaliert werden.

Die Funktion *setMinVelo*

Da unter Umständen sehr viele Sterne angezeigt werden, kann die Darstellung der Geschwindigkeiten durch Pfeile die Visualisierung sehr unübersichtlich machen. Um dem entgegen zu wirken, kann mit Hilfe der Klassenfunktion *setMinVelo* eine Minimalgeschwindigkeit angegeben werden. Bei Partikeln, deren Geschwindigkeit geringer ist als die Minimalgeschwindigkeit, wird der Geschwindigkeitswert in einer Kopie des Geschwindigkeitsvektors zu Null gesetzt. Die Pfeile werden

anhand dieses Vektors skaliert. Daher werden die Geschwindigkeitspfeile unter der Minimalgeschwindigkeit mit dem Wert Null skaliert und infolgedessen nicht angezeigt.

Die Funktion *Switch*

Mittels dieser Funktion kann zwischen den beiden Darstellungsweisen (Kugeln oder Punkte) der Positionen gewählt werden. Je nachdem, welche Darstellung gewählt wird, wird entweder das Objekt *vtkSphereSource* oder *vtkPointSource* in die Visualisierungs-Pipeline eingefügt. Dazu wird das jeweilige Objekt als Quelle des *vtkGlyph3D* angegeben.

Die Funktionen *setResolution, setSphereRadius* und *setSphereRadiusScale*

Die Eigenschaften der Kugeln können mit Hilfe dieser Funktionen beeinflusst werden.
Wie im Konzept bereits beschrieben, bestehen die einzelnen Kugeln aus Polyedern. Mit der Funktion *setResolution* kann die Anzahl der Eckpunkte der Polyeder und somit die Auflösung der Kugeln festgelegt werden. Mit der Auflösung der Kugeln kann der Speicherplatzbedarf und somit die Geschwindigkeit der Darstellung beeinflusst werden.
Die Funktion *setSphereRadius* setzt den Radius der Kugeln auf den angegebenen Wert und mit der Funktion *setSphereRadiusScale* wird das Attribut gewählt, mit dem der Radius der Kugeln skaliert werden soll.

Die Funktion *setLUT*

Die Funktion *setLUT* ist für die Abbildung eines Attributes auf die Farben der Objekte zuständig. Sie wird aufgerufen, wenn der Benutzer ein Attribut für die Farbgebung auswählt.
Die Kugeln bzw. Punkte und die Geschwindigkeitspfeile eines Sternes erhalten die gleiche Farbe. Hierzu wird dem Objekt *vtkPoints* in der jeweiligen Visualisierungs-Pipeline der neue Attributvektor zugewiesen.
Der Funktion wird eine *vtkLookupTable* übergeben. Das ist ein Objekt, welches eine Tabelle enthält, über die skalare Werte auf Farben abgebildet werden können. Es wird der Klasse *vtkPolyDataMapper* zugeordnet, die dann die Abbildung vornimmt.

Die Funktion *setLabel*

Wie im Konzept beschrieben, muss aus der Darstellung hervorgehen, welche Eigenschaft (Attribut) auf die Größe der Kugeln abgebildet ist. Dazu wird in der Funktion *setLabel* eine Legende erzeugt. Die Funktion wird immer dann ausgeführt, wenn sich die Skalierung der Kugeln ändert. Zudem geht aus der Legende noch die Anzahl der Sterne in der Simulation hervor.

Die Funktion *setBoundingBox*

Mit Hilfe dieser Funktion werden die Bounding-Box und die Achsen in die Graphik eingefügt. Um eine Bounding-Box zu erzeugen, wird der Klasse *vtkOutlineFilter* das Objekt *vtkUnstructuredGrid*, welches die Ortskoordinaten aller Sterne enthält, übergeben. *vtkOutlineFilter* berechnet dann die acht Eckpunkte des kleinsten Quaders, der alle Sterne umschließt. Mit Hilfe dieser Punkte werden die Kanten des Quaders erzeugt, welche die Bounding-Box beschreiben. Auch hier wird wieder ein Mapper benötigt, der die Daten in graphische Primitiven umsetzt. Die Visualisierungs-Pipeline wird in Abbildung 6.9 auf der nächsten Seite dargestellt. Neben der Bounding-Box werden auch die Achsen erzeugt. Hierzu wird der Instanz der Klasse *vtkCubeAxesActor2D* ebenfalls das Objekt *vtkUnstructuredGrid* mit den Positionen aller Sterne übergeben. *vtkCubeAxesActor2D* erzeugt auf dieser Grundlage die Achsen mit deren Beschriftung. Die Position der Achsen wird durch

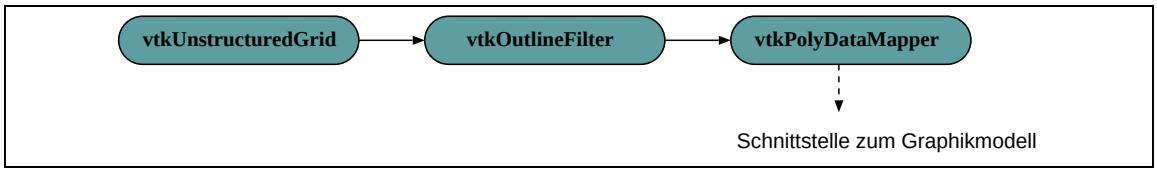


Abbildung 6.9: Visualisierungs-Pipeline für die Bounding-Box

die Bounding-Box festgelegt. In diesem Fall muss kein Mapper instanziiert werden, da die Klasse *vtkCubeAxesActor2D* bereits einen Mapper beinhaltet.

6.3.2 Die Klasse *Tracks*

Diese Klasse ist für die Darstellung der Flugbahnen zuständig. Sie enthält einen Vektor mit den Nummern der Sterne, deren Flugbahnen dargestellt werden sollen, sowie deren Anzahl. Die Funktionen der Klasse werden in den nächsten Abschnitten erläutert.

Der Konstruktor

Der Konstruktor der Klasse *Tracks* wird mit der Anzahl der Sterne in der Simulation aufgerufen. Hier wird dann ein Feld entsprechender Größe erzeugt, in dem später die Partikelnummern gespeichert werden können. Die Anzahl der Partikel, deren Flugbahnen bestimmt werden soll, wird ebenfalls in einer Klassenvariablen festgehalten und kann so von jeder Funktion der Klasse abgefragt werden.

Die Funktion *DisplayTracks*

Die Funktion *DisplayTracks* leitet den Prozess der Darstellung ein. Sie wird mit einer Liste von Partikelnummern und der Kennzahl des Attributs, auf welches die Farben abgebildet werden sollen, aufgerufen. In der Funktion werden diese Daten in den Klassenvariablen abgespeichert. Dann wird die Funktion *refreshTracks* aufgerufen, die im nächsten Abschnitt erläutert wird.

Die Funktion *refreshTracks*

In der Funktion *refreshTracks* wird die Visualisierungs-Pipeline für die Anzeige der Flugbahnen aufgebaut. Im Unterschied zur Visualisierungs-Pipeline der Klasse *scene3d* werden in der Funktion *refreshTracks* die Daten nicht der Klasse *dataReader*, sondern direkt der Klasse *Buffer* entnommen. Hierzu wird über die Klasse *Buffer* die Funktion *nbodybuf_gettrack* benutzt. Diese liefert ein Feld mit den Flugbahndatensätzen. Für alle Flugbahnen wird nun eine Visualisierungs-Pipeline zusammengesetzt (vgl. Abbildung 6.10 auf der nächsten Seite). Die Ortskoordinaten der Sterne und das vom Benutzer ausgewählte Attribut wird aus den Datensätzen herausgefiltert und mit Hilfe der Klasse *vtkPoints* abgespeichert. Es wird ein *vtkUnstructuredGrid* erzeugt, dem die in der Klasse *vtkPolyLine* festgelegte Topologie zugeordnet wird. Es handelt sich hierbei um eine geordnete Liste von Punkten, wobei zwischen je zwei benachbarten Punkten eine Linie gezeichnet wird. Als Geometrie werden die Ortskoordinaten mit Hilfe der *vtkPoints* angegeben. Zum Schluss wird die Pipeline durch ein Objekt der Klasse *vtkDataSetMapper* abgeschlossen.
Die Farben der Tracks werden durch das Attribut bestimmt, welches durch *vtkPoints* in die Visualisierungs-Pipeline eingeht. Die Abbildung des Attributs auf Farben übernimmt der *vtkDataSetMapper.*

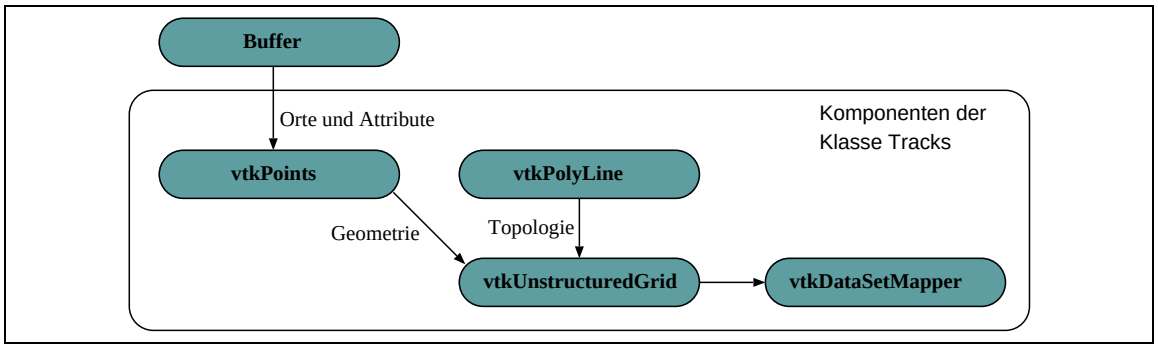


Abbildung 6.10: Visualisierungs-Pipeline der Klasse *Tracks*

Der Funktion *refreshTracks* wird der *Renderer* aus der Klasse *vtk3dWindow* übergeben, so dass das Graphikmodell hier vervollständigt werden kann. Dazu wird für jede Flugbahn ein *vtkActor* erzeugt, der sie repräsentiert. Diese Aktoren werden dem *Renderer* zugeordnet.

6.3.3 Die Klasse *vtk3dWindow*

Die Klasse *vtk3dWindow* ist für die Integration einer VTK-Graphik in ein Qt-Widget zuständig. Hierzu wird eine Instanz von der Klasse *vtkRenderWindow* erzeugt, die dann mit Hilfe der Schnittstellenbibliothek *vtk_qt1.7Beta* [L13] in ein Qt-Objekt umgewandelt wird (vgl. Abbildung 6.11).

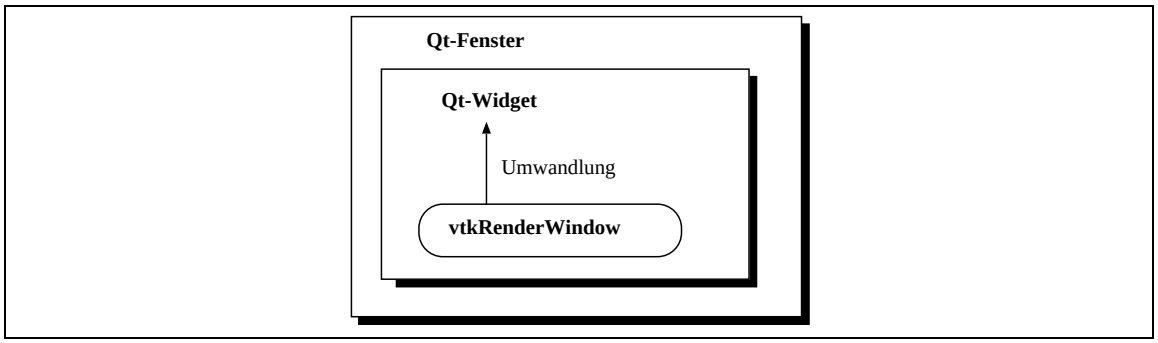


Abbildung 6.11: Einbettung der VTK-Graphik in ein Qt-Fenster Die Instanz der Klasse *vtkRenderWindow* wird in ein Qt-Widget umgewandelt. Dieses erhält als übergeordnetes Widget ein Qt-Fenster, durch welches die Position der Anzeige in der graphischen Oberfläche bestimmt ist.

Zusätzlich wird in dieser Klasse die Graphikschnittstelle für die 3-dimensionale Visualisierung bereitgestellt. Hierzu werden die Objekte *vtkRenderWindow* und *vtkRenderer* miteinander verknüpft. Durch den Einsatz der Klasse *vtkRenderWindowInteractor* wird die Interaktion mit der Graphik ermöglicht. Der Benutzer kann das 3-dimensionale Bild zum Beispiel drehen oder hinein- und herauszoomen.
Des Weiteren verwaltet die Klasse *vtk3dWindow* je eine Instanz der Klassen *scene3d* und *Tracks*, die in den Kapiteln 6.3.1 und 6.3.2 beschrieben wurden. Zwei Farblegenden werden in die Graphik integriert, damit der Benutzer die Farben der Sterne und der Flugbahnen interpretieren kann (vgl. Abbildung 6.12 auf der nächsten Seite).

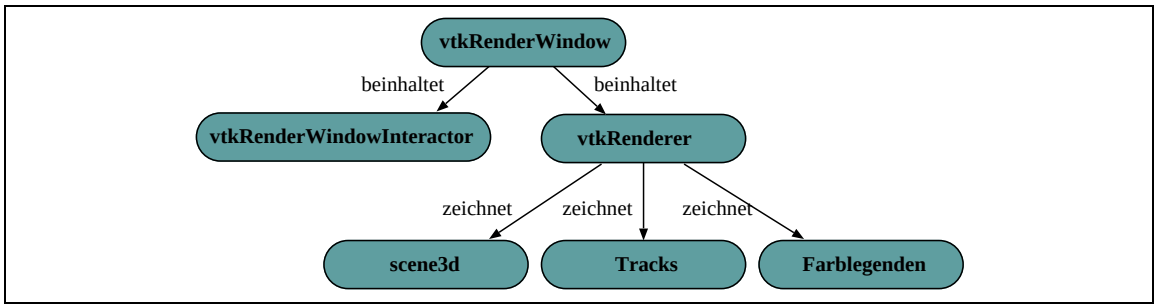


Abbildung 6.12: Beziehungen zwischen den für die Graphikschnittstelle relevanten Bestandteilen der Klasse *vtk3dWindow*

6.4 2-dimensionale Darstellung der Daten

Die 2-dimensionale Darstellung wird durch die Klassen *vtk2dWindow* und *scene2d* realisiert. Zunächst wird die Klasse *scene2d* vorgestellt, da diese ein Bestandteil der Klasse *vtk2dWindow* ist.

6.4.1 Die Klasse *scene2d*

In der Klasse *scene2d* wird die Visualisierungs-Pipeline für die Erzeugung der 2-dimensionalen Darstellung implementiert. Hierzu werden zwei Vektoren benötigt, welche die vom Benutzer ausgewählten Kompontenten aller Sterne zu einem Zeitpunkt beinhalten. Diese werden von der Klasse *DataReader* zur Verfügung gestellt. Die Klasse *DataReader* dient also, bezogen auf das Visualisierungsmodell, als *Quelle* (vgl. Abbildung 2.3).
In einem *vtkDataObject* werden die Daten zwischengespeichert und an die Klasse *vtkXYPlotActor* weitergereicht. Hier werden dann die folgenden Komponenten erzeugt:

- ein Koordinatenkreuz inklusive Achsenbeschriftung und
- ein Graph, in diesem Fall eine Punktwolke.

Graphik- und Visualisierungsmodell werden hier nicht strikt getrennt. Die Klasse *vtkXYPlotActor* verwendet sowohl einen *Mapper*, der den Datenfluss in der Visualisierungs-Pipeline abschließt, als auch Teile des Graphikmodells. Diese werden in der Klasse *vtk2dWindow*, die im nächsten Kapitel beschrieben wird, aufgegriffen und dort in das Graphikmodell integriert (vgl. Abbildung 6.13).

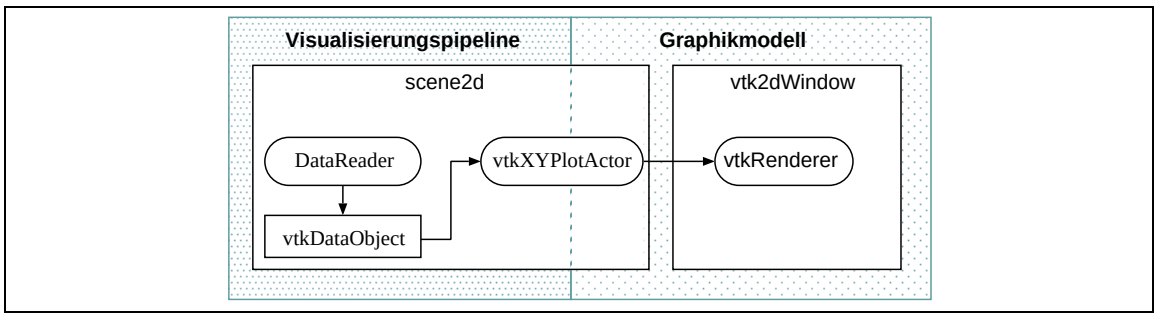


Abbildung 6.13: Einordnung der Klasse *scene2d* in das Visualisierungs- bzw. Graphikmodell

Die Klasse besitzt einen Konstruktor, in dem die benötigten Objekte und Variablen angelegt und initialisiert werden, und eine Funktion *setNewScene()*. In dieser Funktion werden neue Daten von der Klasse *DataReader* eingelesen und die Graphik entsprechend aktualisiert.

6.4.2 Die Klasse *vtk2dWindow*

Die Klasse *vtk2dWindow* ist ähnlich aufgebaut wie die Klasse *vtk3dWindow* (vgl. Kapitel 6.3.3). Die Integration der Graphik in ein Qt-Widget erfolgt analog.
Es wird eine Graphikschnittstelle für die 2-dimensionale Visualisierung bereitgestellt. Ausserdem beinhaltet die Klasse *vtk2dWindow* eine Instanz der Klasse *scene2d*, in der, wie in Kapitel 6.4.1 beschrieben, die Visualisierungs-Pipeline aufgebaut wird (vgl. Abbildung 6.14).

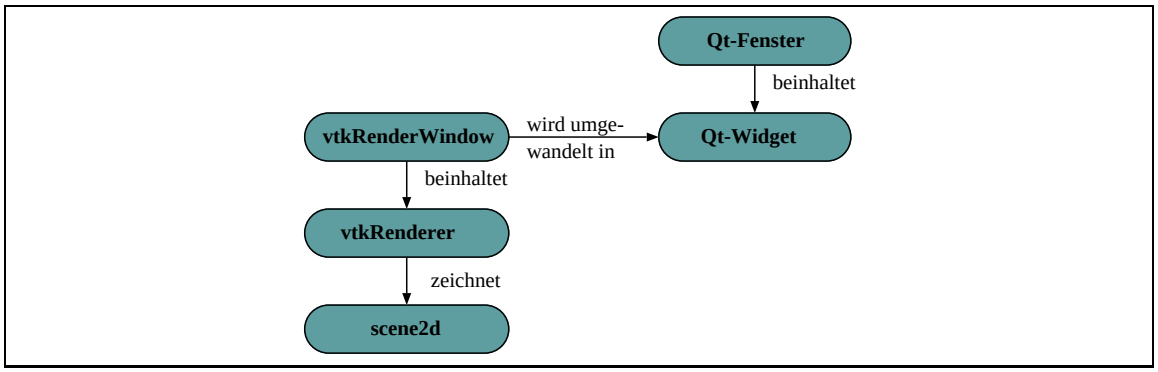


Abbildung 6.14: Beziehungen zwischen den Bestandteilen der Klasse *vtk2dWindow*

Nachfolgend werden die Funktionen der Klasse *vtk2dWindow* vorgestellt.

Der Konstruktor

Im Konstruktor der Klasse *vtk2dWindow* werden alle in Abbildung 6.14 aufgeführten Objekte, ausgenommen das Objekt *scene2d*, erzeugt und in der oben dargestellten Form miteinander verbunden. Das Objekt *scene2d* kann erst im weiteren Verlauf des Programmes erzeugt werden, da vorausgesetzt wird, dass bereits Simulationsdaten vorliegen.

Die Funktion *refresh()*

Diese Funktion wird immer dann aufgerufen, wenn das Objekt *DataReader* neue Datensätze eingelesen hat oder der Benutzer ein anderes Attribut für die 2-dimensionale Darstellung auswählt.
In jedem Fall sind bei Eintritt in diese Funktion Datensätze vorhanden, so dass das Objekt *scene2d*, falls es nicht bereits angelegt wurde, jetzt erzeugt werden kann. Anschließend wird es durch den Aufruf der Funktion *setNewScene()* dazu veranlasst, der Klasse *DataReader* andere bzw. aktuellere Datensätze zu entnehmen und diese darzustellen.

Die Funktion *resizeEvent()*

Die Klasse *vtk2dWindow* wird von *QWidget* abgeleitet. *QWidget* beinhaltet die Funktion *resizeEvent()*, die in der abgeleiteten Klasse überschrieben wird. Sowohl das Qt-Fenster als auch das Objekt *vtkRenderWindow* werden so dynamisch an die Größe des übergeordneten Qt-Fensters angepasst.

6.5 Bereitstellung der Daten

Um den Zugriff auf die Daten des Zwischenspeichers zu erleichtern, wird eine Klasse erstellt, welche die Datensätze eines bestimmten Simulationszeitpunktes bereithält. In der Klasse werden also

Daten aus dem Zwischenspeicher eingelesen und in eine für die Visualisierung sinnvolle Form gebracht.

6.5.1 Die Klasse *DataReader*

Die Klasse beinhaltet Datenstrukturen für die Simulationsdaten jeweils eines Simulationszeitpunktes. Diese werden im Konstruktor der Klasse *DataReader,* der im folgenden Abschnitt beschrieben wird, erzeugt. In den darauf folgenden Abschnitten werden weitere Funktionen der Klasse erläutert, mit deren Hilfe die Daten einerseits aus dem Zwischenspeicher eingelesen und andererseits von anderen Modulen abgefragt werden können.

Der Konstruktor

Für die Darstellung der Simulationsdaten sind oft Arrays mit den entsprechenden Attributdaten erforderlich. Im Konstruktor werden, wie in Abbildung 6.15 dargestellt, Instanzen der Klassen *vtkFieldData* und *vtkFloatArray* erzeugt, wobei *vtkFloatArray* jeweils ein Datenfeld eines Attributes beinhaltet. Die Arrays aller Attribute werden mit Hilfe der Klasse *vtkFieldData* zu einem Array von Arrays verknüpft. Mehrdimensionale Attribute (wie zum Beispiel die Geschwindigkeit) werden genau wie eindimensionale Attribute (zum Beispiel die Identifikationsnummer) durch ein *vtkFloatArray* repräsentiert. Die Dimension eines *vtkFloatArray* kann durch Funktionen der Klasse festgelegt und auch abgefragt werden. Durch diese Strukturierung der Daten kann eine Abfrage der Attribute generisch durch Angabe einer Attributkennziffer erfolgen.

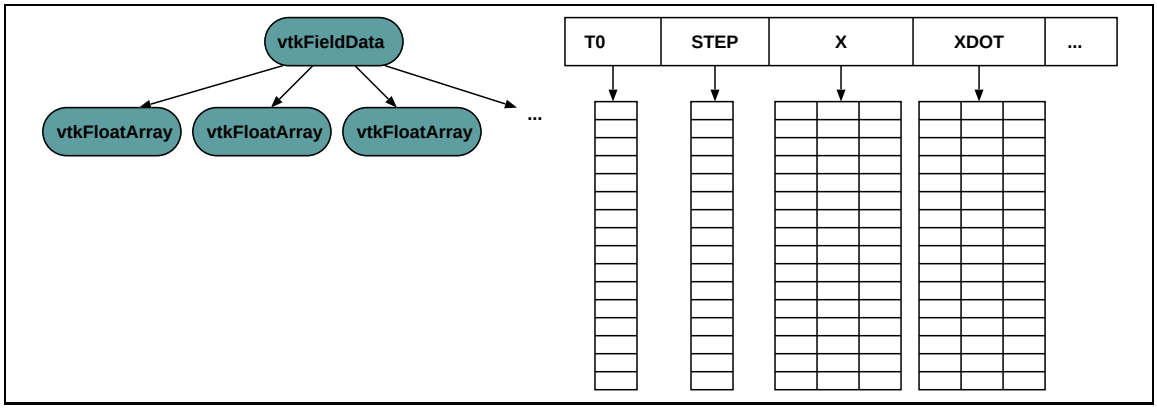


Abbildung 6.15: Datenstrukturen der Klasse *DataReader*: Es wird ein Array erzeugt, welches die Arrays mit den Attributwerten der Simulationsdaten zum aktuell betrachteten Zeitpunkt beinhaltet. Die Attribut-Arrays können unterschiedliche Dimensionen haben.

Da für die 3-dimensionale Darstellung der Daten in jedem Fall die Ortskoordinaten benötigt werden, werden diese in der Form von *vtkPoints* zur Verfügung gestellt.

Die Funktion *getDataFromBuffer*

In dieser Funktion werden die Simulationsdaten, die zum aktuell betrachteten Simulationszeitpunkt gehören, aus dem Zwischenspeicher in die Datenstrukturen der Klasse *DataReader* eingelesen. Der Zeitpunkt wird der Funktion als Parameter übergeben.

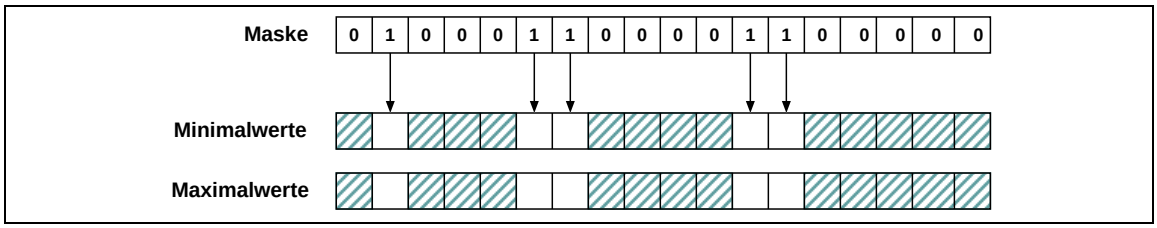


Abbildung 6.16: Eingabeparameter der Funktion *getIds*: Jedem Attribut ist ein Index in den Arrays Minimalwert, Maximalwert und Maske zugeordnet. Der Wert in der Array-Maske bestimmt, ob der Minimal- und Maximalwert auf das jeweilige Attribut angewendet werden soll. Auf diese Weise können alle möglichen Einschränkungen miteinander kombiniert werden.

Die Funktionen *getData, getPointData* und *get2DPoints*

Diese Funktionen liefern die entsprechenden Datenstrukturen für die Visualisierung. Die Funktion *getData* wird mit einer Attributkennziffer aufgerufen und liefert dann das entsprechende Array als *vtkFloatArray* aus der Struktur *vtkFieldData* zurück. Diese Datenstrukturen werden für die Skalierung und Farbgebung von Objekten nach bestimmten Attributen benötigt.

Die Funktion *getVelo*

Diese Funktion vergleicht den Betrag der Geschwindigkeit jedes Partikels mit einem Minimalwert, der ihr als Parameter übergeben wird. Ist der Betrag kleiner als die Minimalgeschwindigkeit, so werden die Geschwindigkeitskomponenten dieses Sterns in einer Kopie des Geschwindigkeitsvektors zu Null gesetzt. *getVelo* liefert das so modifizierte Attributarray zurück.

Die Funktion *getIds*

Mit Hilfe der Funktion *getIDs* können die Identifikationsnummern von Partikeln mit bestimmten Eigenschaften herausgefiltert werden. Benötigt wird diese Funktion, wenn für die Anzeige von Flugbahnen die Partikel nicht anhand ihrer Identifikationsnummer, sondern auf Grund bestimmter Eigenschaften ausgewählt werden sollen. Hierzu werden der Funktion Arrays mit den Minimal- und Maximalkriterien für alle Attribute sowie eine Maske übergeben. Es müssen nicht zu jedem Attribut Minimal- und Maximalkriterien angegeben werden. In der Maske wird festgelegt, auf welche Attribute die Kriterien anzuwenden sind (vgl. Abbildung 6.16). *getIds* liefert die Identifikationsnummern der Partikel, die diese Kriterien erfüllen, in einem Ausgabefeld zurück.

Die Funktion *getLookupTable*

getLookupTable liefert eine *vtkLookupTable*, über welche die Abbildung der Attribute auf die Farben erfolgt. Die Funktion erhält eine Attributkennziffer, anhand derer sie den Minimal- und Maximalwert des gewählten Attributs der Tabelle zuordnet.

6.6 Die graphische Benutzeroberfläche

Die graphische Benutzeroberfläche wird durch die Klasse *Mainwindow* realisiert. Sie erzeugt das Hauptfenster mit Objekten zur Interaktion zwischen Benutzer und Visualisierung. Darüber hinaus stellt die Klasse *Mainwindow* das Bindeglied zwischen den in den vorherigen Kapiteln beschriebenen Modulen von *xnbody* dar.

6.6.1 Die Klasse *Mainwindow*

Die graphische Oberfläche wird durch eine Hierarchie von Qt-Objekten (Widgets) erzeugt. In der folgenden Tabelle werden die wichtigsten hierzu erforderlichen Objekte und ihre Funktion aufgeführt:

Widget	Funktion
QMainWindow	Dieses Objekt erzeugt ein Hauptfenster für eine Anwendung und stellt eine Menüleiste, eine Werkzeugleiste sowie eine Statusleiste zur Verfügung.
QHBox bzw. *QVBox*	Diese Widgets sind nicht sichtbare Objekte, die für das Layout benötigt werden. Sie können mehrere andere Widgets beinhalten, die sie dann horizontal bzw. vertikal anordnen.
QGroupBox	*QGroupBox* ist ein Widget, welches mehrere andere Widgets gruppiert. Die untergeordneten (Child-) Widgets können horizontal oder vertikal angeordnet werden. Es kann ein Rahmen um die Child-Widgets und ein Titel der Gruppe angezeigt werden.
QToolBox	*QToolBox* ermöglicht die platzsparende Anzeige von mehreren Widgets. Es wird nur ein Widget in voller Größe angezeigt während für die anderen eine Schaltfläche eingeblendet wird. Durch Anklicken dieser Schaltflächen kann dieses Widget ausgewählt werden.
QLabel	Mit Hilfe dieses Widgets lassen sich Textzeilen in der Oberfläche anzeigen.
QLineEdit	Dieses Widget stellt dem Benutzer eine Eingabezeile zur Verfügung.
QPushButton	Ein *QPushButton* ist ein Schalter, der bei Anklicken ein Signal aussendet. Er kann mit einem Text beschriftet oder einer Abbildung versehen werden.
QSlider bzw. *QSpinBox*	Diese Widgets eignen sich als Einstellmöglichkeit für einen Wert, der in einem bestimmten Bereich liegt.
QComboBox	Hiermit können Elemente aus einer Liste ausgewählt werden.
QProgressBar	Eine *QProgressBar* wird benutzt, um den Fortschritt eines Prozesses oder eine Auslastung (in Prozent) zu visualisieren.
QTabDialog	Mit Hilfe dieser Klasse können vorgefertigte Dialoge erzeugt werden, die durch Ableiten an die entsprechenden Anforderungen angepasst werden können.

In Abbildung 6.17 auf der nächsten Seite wird die hierarchische Anordnung der Widgets angedeutet. Um ein Widget einem anderen Widget unterzuordnen, wird bei der Instanziierung des Child-Widgets das ParentWidget im Konstruktor angegeben.

Neben den Objekten für den Aufbau der graphischen Oberfläche (wozu auch die 2- und 3-dimensionalen Darstellungen der Simulationsdaten gehören) werden im Konstruktor der Klasse *Mainwindow* die oben beschriebenen Komponenten *Buffer*, *DataReader* und *Connection* erzeugt. Um über die Schalter und Eingabefelder der graphischen Oberfläche mit diesen Komponenten interagieren zu können, werden die Signale der Qt-Objekte ebenso wie die der Komponenten mit entsprechenden Slots verbunden.
Nachfolgend werden den oben genannten Komponenten sowie auch den Komponenten *vtk3dWindow*

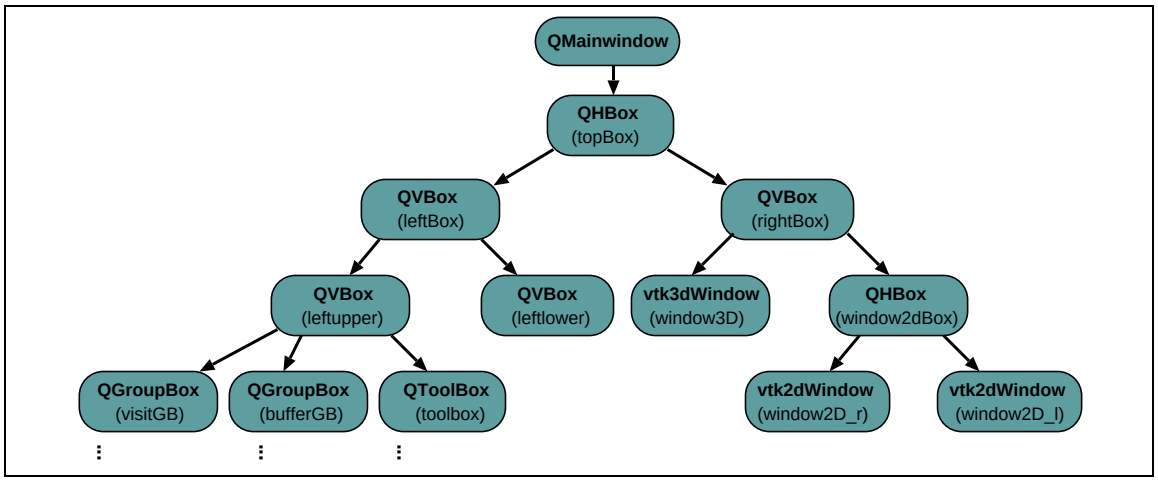


Abbildung 6.17: Hierarchie der Qt-Widgets in der graphischen Oberfläche: Das Erscheinungsbild der graphischen Oberfläche wird durch eine Hierarchie, welche hier nicht vollständig dargestellt ist, bestimmt. Neben „reinen“ Qt-Widgets werden auch die Klassen *vtk3dWindow* und *vtk2dWindow* in die Hierarchie eingegliedert und erhalten so ihren Platz in der graphischen Oberfläche.

und *vtk2dWindow* die Elemente der graphischen Oberfläche sowie die dazugehörigen Funktionen zugeordnet.

Buffer

Die graphische Oberfläche hat in Bezug auf die Komponente *Buffer* rein informativen Charakter. Sie zeigt mit Hilfe eines *QLabel*s die Größe des verwendeten Zwischenspeichers an. Die Auslastung wird mittels einer *QProgressBar* dargestellt. Diese wird immer dann aktualisiert, wenn von der Klasse *Buffer* das Signal „*newDataInBuffer*“ ausgesendet wird.

Connection

Für die Komponente Connection werden mehrere Eingabefelder (*QLineEdit*) zur Angabe der für den Verbindungsaufbau relevanten Informationen bereitgestellt. Des Weiteren wird ein *QPushButton* erzeugt, dessen Signal „*clicked*“ mit dem Slot „*slotConnection*“ verbunden wird. Diese Funktion wiederum sorgt dann dafür, dass im Modul *Connection* die entsprechenden Funktionen ausgeführt werden.
Ein weiterer *QPushButton* ist mit dem Slot „*slotTunnel*“ verbunden, der in der Klasse *Connection* dafür sorgt, dass die VISIT-Verbindung über einen ssh-Tunnel aufgebaut wird.
Für das Steering werden ebenfalls mehrere Eingabefelder erzeugt. Wenn eine Steering-Nachricht an die Simulation geschickt werden soll sendet die Klasse *Connection* das Signal „*steering*“ aus, welches in der Klasse *Mainwindow* mit dem Slot „*slotReadSteeringArgs*“ verbunden ist. In diesem Slot werden die Inhalte der Eingabefelder abgefragt und an die Klasse *Connection* weitergeleitet.

DataReader

Der Benutzer hat mehrere Möglichkeiten, einen Simulationszeitpunkt zur Anzeige auszuwählen. Er kann mit Hilfe eines *QSlider*s den Zeitpunkt mit Hilfe der Maus einstellen, aber auch in einem *QLineEdit* über die Tastatur einen Zeitpunkt explizit eingeben. Des Weiteren werden Schalter zur Navigation innerhalb des Zwischenspeichers zur Verfügung gestellt. Sobald sich der anzuzeigende Simulationszeitpunkt ändert, wird die Klasse *DataReader* veranlasst, die Daten zu diesem Zeitpunkt einzulesen.

vtk3dWindow

Sowohl in der Menüleiste als auch in der Werkzeugleiste des Hauptfensters werden Einstellmöglichkeiten für die 3-dimensionale Darstellung angeboten.
So können zum Beispiel zur Darstellung der Sterne Kugeln oder Punkte gewählt werden. Wird die Schaltfläche für die Kugeln angeklickt, so wird ein Dialog geöffnet, mit dessen Hilfe sich die Eigenschaften Größe, Auflösung und Skalierung der Kugeln einstellen lassen. Bei Bestätigung der Eingaben werden diese an das Modul *vtk3dWindow* weitergeleitet.
Auf die gleiche Weise wird mit der Auswahl von Partikeln zur Anzeige von Flugbahnen verfahren.

vtk2dWindow

Für die Interaktion mit der 2-dimensionalen Darstellung der Simulationsdaten werden für beide Instanzen von *vtk2dWindow* je zwei Objekte von *QComboBox* erzeugt. Diese enthalten als Elemente zur Auswahl die Orts- und Geschwindigkeitskomponenten. Ändert sich das ausgewählte Element, so wird das Signal *activated* gesendet, welches mit einer Funktion verbunden ist, die in der Klasse *vtk2dWindow* die entsprechenden Änderungen hervorruft.

Im folgenden Kapitel wird auf die Online-Visualisierung *xnbody*, die durch die oben beschriebene Implementierung entstand, eingegangen.

Kapitel 7

Die Online-Visualisierung *xnbody*

In diesem Kapitel wird die im Rahmen dieser Arbeit erstellte Online-Visualisierung *xnbody* vorgestellt. Hierzu wird zunächst die graphische Oberfläche beschrieben. Anhand eines Beispiels wird dann die Visualisierung des dynamischen Verhaltens eines Sternhaufens gezeigt. Zuletzt werden die Möglichkeiten der Darstellung von Partikeleigenschaften und Flugbahnen demonstriert.

7.1 Die graphische Oberfläche

Die in dieser Arbeit erstellte graphische Oberfläche von *xnbody* ist in Abbildung 7.1 auf der nächsten Seite dargestellt. Am oberen Rand sind eine Menüleiste sowie eine Werkzeugleiste angebracht. In der Menüleiste befinden sich die Schaltflächen *Appearance, View, Tracks* und *About.* Über die Schaltfläche *Appearance* lassen sich die Darstellungsart der Sterne (Kugeln oder Punkte) und das Attribut, auf welches die Farben der Sterne abgebildet werden, einstellen. Des Weiteren kann hier die *Bounding Box* ein- und ausgeschaltet werden. Der Menüeintrag *View* beinhaltet einen Schalter zur Repositionierung der „Kamera“. Wird dieser Schalter betätigt, so wird die Visualisierung so ausgerichtet, dass alle Sterne sichtbar werden. Über die Schaltfläche *Tracks* wird ein Dialog geöffnet, mit dessen Hilfe Gruppen von Sternen ausgewählt werden können, deren Flugbahnen angezeigt werden sollen. Des Weiteren kann auch hier ein Attribut ausgewählt werden, auf welches die Farben der Polygonzüge abgebildet werden. In dem Menüeintrag *About* werden Informationen zu VISIT und *xnbody*, wie zum Beispiel die Versionsnummer, bereitgestellt.
Die Werkzeugleiste bietet ebenfalls Schalter zum Umschalten zwischen der Darstellung durch Kugeln oder Punkte sowie zur Repositionierung der Kamera. Auch die Informationen über VISIT können mit Hilfe eines Elements in der Werkzeugleiste angezeigt werden. Darüber hinaus ist hier eine *Spinbox* zur Kontrolle der Minimalgeschwindigkeit für die Geschwindigkeitsanzeige in die Werkzeugleiste integriert. Über einen weiteren Schalter kann die Anwendung geschlossen werden.

Unterhalb von Menü- und Werkzeugleiste befinden sich links der Kontrollbereich und rechts die Anzeigefenster von *xnbody*. Der Kontrollbereich wiederum ist unterteilt in mehrere Abschnitte:

Visit In den hier angezeigten Eingabefeldern kann der Benutzer die für den Verbindungsaufbau notwendigen Angaben machen. Des Weiteren befindet sich hier eine Schaltfläche, über die der VISIT-Server gestartet werden kann. Diese ist zu Anfang rot eingefärbt, wird bei Betätigung zunächst gelb und bei Zustandekommen einer Verbindung mit der Simulation grün.

Buffer Im Abschnitt *Buffer* können Größe und Auslastung des Zwischenspeichers abgelesen werden.

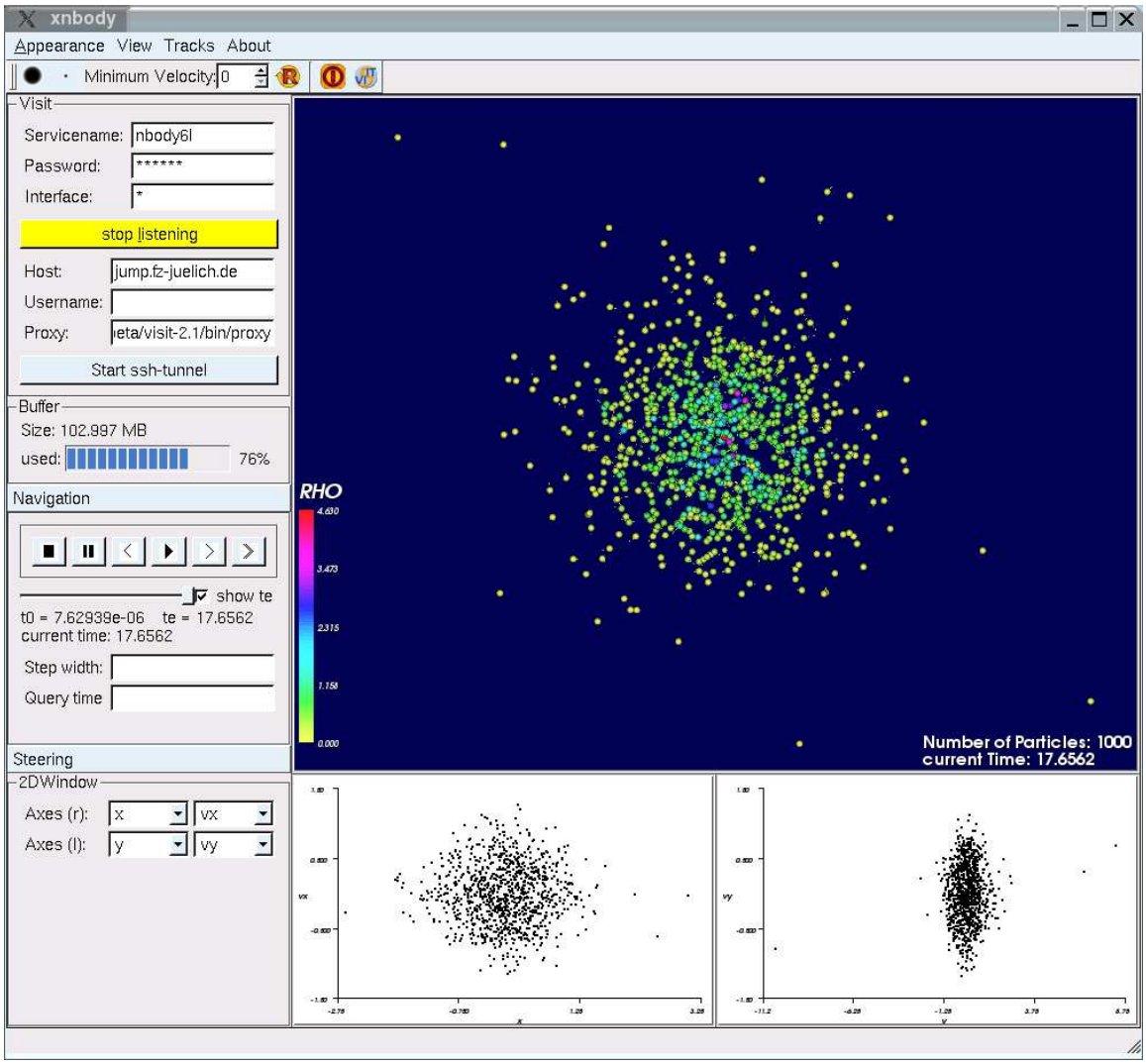


Abbildung 7.1: Die graphische Benutzeroberfläche von *xnbody*: Die Oberfläche ist aufgeteilt in einen Kontrollbereich und einen Bereich für die graphischen Darstellungen der Simulationsdaten. Im Kontrollbereich können verschiedene Einstellungen vorgenommen werden, zum Beispiel die Verbindung mit der Simulation betreffend oder auch die Navigation innerhalb des Zwischenspeichers. Visualisiert wird die Simulation eines Sternhaufens mit 1000 Sternen zu einem relativ frühen Simulationszeitpunkt, weshalb sich die Sterne noch nahezu in ihrer kugelförmigen Anordnung befinden.

Navigation Hier werden Schaltflächen und Eingabefelder zur Auswahl eines im Zwischenspeicher vorliegenden Simulationszeitpunktes angezeigt.

Steering Dieser Abschnitt besteht aus Eingabefeldern, in die Steering-Parameter verändert werden können. Diese werden ausgelesen und an die Simulation geschickt, sobald diese ihre Bereitschaft zur Annahme von Steering-Parametern signalisiert.

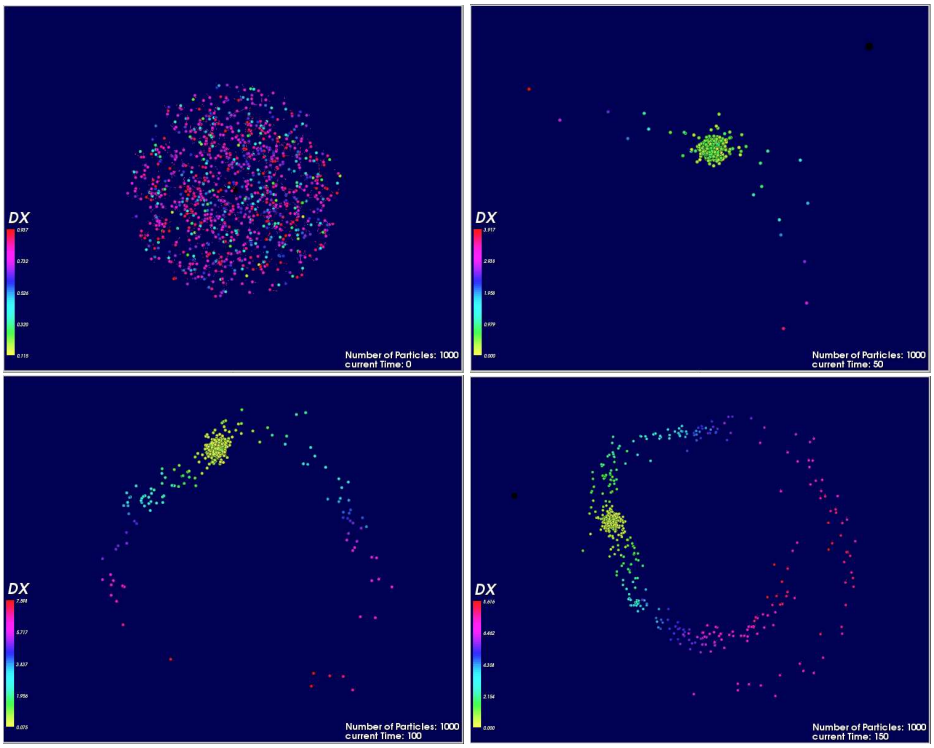
Aus Gründen der Kompaktheit werden die Abschnitte Navigation und Steering nicht zeitgleich in voller Größe angezeigt. Durch Anklicken der entsprechenden Schaltfläche kann der Benutzer zwischen den beiden Abschnitten wählen. In Abbildung 7.1 wird der Abschnitt Steering nicht angezeigt.

2DWindow Hier können die Geschwindigkeits- und Ortskomponten für die 2-dimensionalen Darstellungen ausgewählt werden.

Rechts neben dem Kontrollbereich werden die Simulationsdaten dargestellt. Im Fenster für die 3-dimensionale Darstellung werden neben der Visualisierung der Simulationsdaten auch die Anzahl der Sterne sowie der Simulationszeitpunkt angezeigt. Der Benutzer kann mit Hilfe der Maus die Ansicht steuern. Hält er die rechte Maustaste gedrückt, so kann er durch vertikale Bewegungen in die Darstellung hinein- bzw. herauszoomen. Wird die mittlere Maustaste gedrückt gehalten, können durch Mausbewegung die Objekte innerhalb des Fensters verschoben werden. Mit Hilfe der linken Maustaste können die Objekte gedreht und von allen Seiten betrachtet werden.

7.2 Demonstration eines Simulationsverlaufs

In Abbildung 7.2 auf der nächsten Seite ist eine Serie von 6 Ausschnitten aus der Oberfläche von *xnbody* zu sehen, die den Verlauf einer Simulation mit 1000 Partikeln zeigt.
Im ersten Ausschnitt ist die Startverteilung der Positionen und Geschwindigkeiten der Partikel abgebildet. Verglichen mit den übrigen Ausschnitten wurden die Partikel hier sehr nahe herangeholt. Die Partikel werden durch Kugeln dargestellt, die anhand ihrer Geschwindigkeiten eingefärbt sind. In einem gewissen Abstand zu dem Kugelsternhaufen wird ein schwarzes Loch simuliert, welches den Sternhaufen umkreist. In der Realität ist dieser Sachverhalt jedoch genau umgekehrt, der Kugelsternhaufen umkreist das Schwarze Loch. Mit Hilfe einer Koordinatentransformation kann jedoch gezeigt werden, dass beide Modelle äquivalent zueinander sind.
Der zweite Ausschnitt zeigt die Partikel zum Simulationszeitpunkt $t = 50$. Hier entfernen sich unter dem Einfluss des schwarzen Loches bereits einige Sterne aus der kugelförmigen Anordnung und bilden Ansätze von Spiralarmen. Die weiteren Ausschnitte zeigen in Zeitschritten von jeweils 50 Zeiteinheiten die Entwicklung des Kugelsternhaufens zu einer Spirale. Die Kugeln sind entsprechend ihrer Geschwindigkeit (*DX*) eingefärbt. Anhand dieser Farbgebung der einzelnen Kugeln wird deutlich, dass sich die Partikel, die dem schwarzen Loch näher sind, relativ langam bewegen und daher im Bereich gelb bis grün der Farbskala liegen. Entferntere Partikel haben in der gleichen Zeit einen längeren Weg zurückzulegen und sind aufgrund ihrer höheren Geschwindigkeit im oberen Bereich der Farbskala (blau bis rot) angesiedelt.



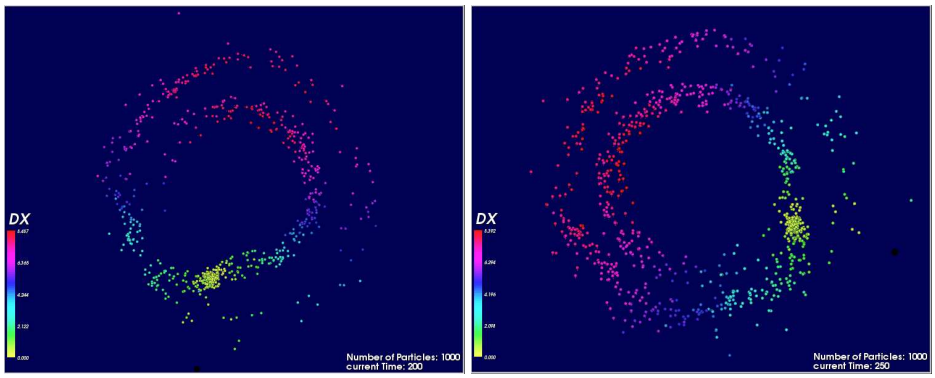


Abbildung 7.2: Aufnahmen der Visualisierung zu einer Simulation mit 1000 Partikeln: Über einen Zeitraum von $t = 0$ bis $t = 250$ sind die Simulationsdaten in Zeitschritten von 50 Zeiteinheiten dargestellt.

7.3 Darstellung von Partikeleigenschaften

Wie im Konzept bereits beschrieben, können neben den Positionen der Partikel auch verschiedene Attribute dargestellt werden. Die Attributwerte können über die Skalierung (im Fall der Darstellung der Partikel durch Kugeln) oder Farbgebung der Objekte, die die Partikel repräsentieren, angezeigt werden.

7.3.1 Farben

In Abbildung 7.3 auf der nächsten Seite wird die Darstellung von Partikelattributen mit Hilfe von Farben anhand eines Beispiels gezeigt. In der Abbildung ist ein Datensatz mit 1000 Partikeln zu einem relativ späten Simulationszeitpunkt $t \approx 1000$ dargestellt. Die Sterne liegen bereits gleichmäßig verteilt auf einem Ring. Zur Darstellung der Partikel wurden Kugeln verwendet. Im rechten Bild wurde als Attribut für die Abbildung auf Farben die Dichte (*RHO*) ausgewählt, während im linken Bild die Geschwindigkeit (*DX*) durch die Farben dargestellt ist. Das gewählte Attribut kann jeweils an der Überschrift der Farblegende links im Bild abgelesen werden. Auf der linken Abbildung wird wiederum deutlich, dass sich die Partikel auf der dem schwarzen Loch zugewandten Seite des Sternhaufens langsamer bewegen als die gegenüberliegenden. Das rechte Bild zeigt, dass die Partikel auf der Innenseite des Ringes eine höhere Dichte haben als an der Außenseite.

7.3.2 Skalierung

Die Darstellung von Partikeleigenschaften durch die Skalierung der Kugeln wird in Abbildung 7.4 auf der nächsten Seite gezeigt. In der Abbildung ist ein Datensatz mit 1000 Partikeln zum Zeitpunkt $t = 249$ zu sehen. Jeweils im Bild unten rechts ist der Legende zu entnehmen, welches Attribut gerade durch die Skalierung der Kugelradien dargestellt wird. Im linken Bild handelt es sich dabei um die Geschwindigkeiten der Partikel. Da die Farben der Kugeln ebenfalls auf die Geschwindigkeiten abgebildet wurden, zeigt das rechte Bild, dass die roten Kugeln (in diesem Fall also die Sterne mit der höchsten Geschwindigkeit) auch die größten Radien besitzen. Im Bild rechts wird die Schrittweite (*STEP*) auf die Radien der Kugeln abgebildet.

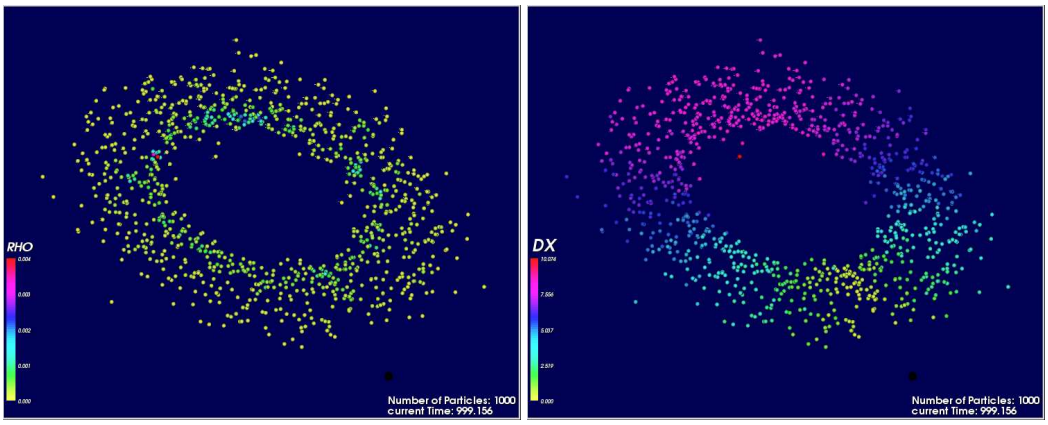


Abbildung 7.3: Darstellung von Attributwerten über die Farbgebung: Die Abbildung zeigt die Visualisierung zu einer Simulation mit 1000 Sternen, wobei die Kugeln, welche die Sterne darstellen, im linken Teil der Abbildung nach dem Attribut *RHO* eingefärbt wurden, während im rechten Teil die Farben auf die Geschwindigkeiten abgebildet wurden. Es wird deutlich, dass die Dichte (*RHO*) bis auf einige Ausreißer bei allen Sternen in etwa gleichermaßen niedrig ist, da die Farben der Kugeln im unteren Bereich der Farbskala liegen. Zum inneren Rand des Ringes hin ist ein leichter Anstieg zu verzeichnen. Im rechten Bild kann man die Geschwindigkeitsverteilung im Sternhaufen ablesen: die Sterne auf der dem schwarzen Loch abgewandten Seite haben die höchste Geschwindigkeit.

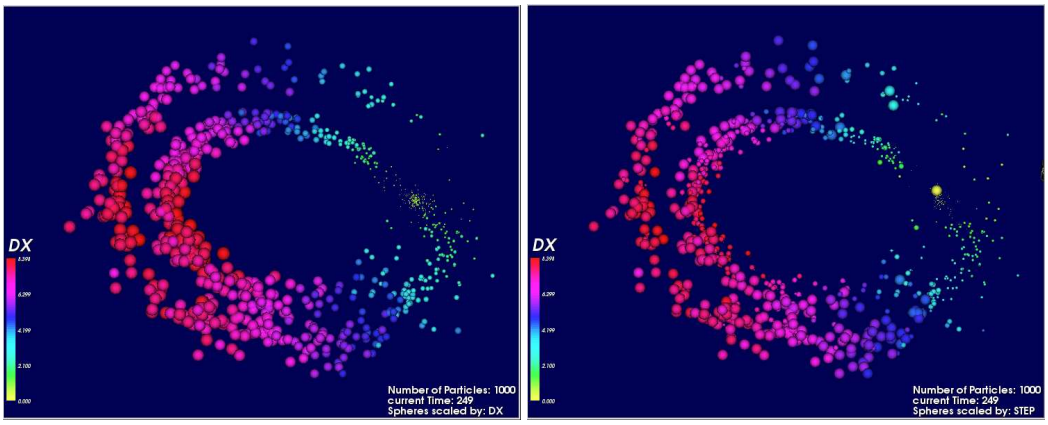


Abbildung 7.4: Darstellung von Attributwerten über die Skalierung der Kugeln: Die Abbildungen zeigen die Simulation von 1000 Partikeln zum Simulationszeitpunkt $t = 249$. Im linken Bild wurden die Radien der Kugeln mit der Geschwindigkeit skaliert, rechts mit der Schrittweite. Es wird deutlich, dass sowohl Schrittweite als auch Geschwindigkeit auf einer Seite des Sternhaufens wesentlich größer sind als auf der anderen Seite.

7.4 Darstellung von Flugbahnen

Die Flugbahn eines Partikels wird durch einen Polygonzug durch die zeitabhängigen Ortskoordinaten des Partikels dargestellt. Die Farbe an einem Punkt in diesem Polygonzug entspricht dem Wert des vom Benutzer ausgewählten Attributes an diesem Punkt. Auf den Strecken dazwischen wird ein Farbverlauf erzeugt. Bei der Farbgebung der Polygone können neben dem Attribut auch verschiedene Grenzen angegeben werden, außerhalb derer die Werte auf die Grenzwerte abgebildet werden. Hierdurch können bestimmte Eigenschaften deutlicher herausgestellt werden.

In Abbildung 7.5 auf der nächsten Seite ist neben dem Datensatz mit 1000 Partikeln auch die Flugbahn des Partikels mit der Nummer 750 dargestellt. In beiden Bildern wird die Flugbahn anhand der

Geschwindigkeit eingefärbt. Im linken Bild wurden als Grenzen das Minimum bzw. Maximum der Geschwindigkeit aller Partikel zum Simulationszeitpunkt $t = 990$ gewählt. Rechts wurde dagegen die Minimal- bzw. Maximalwerte des gesamten Zwischenspeichers als Grundlage verwendet. Wie schon im Kapitel 4.4 angesprochen, haben beide Varianten sowohl Vor- als auch Nachteile. Wird ein zu kleines Intervall gewählt, können die Werte außerhalb nicht korrekt dargestellt werden. Bei einem zu großem Intervall besteht die Gefahr, dass die Darstellung nicht mehr differenziert genug ist und nahe beieinander liegende Werte auch sehr ähnliche Farben erhalten.

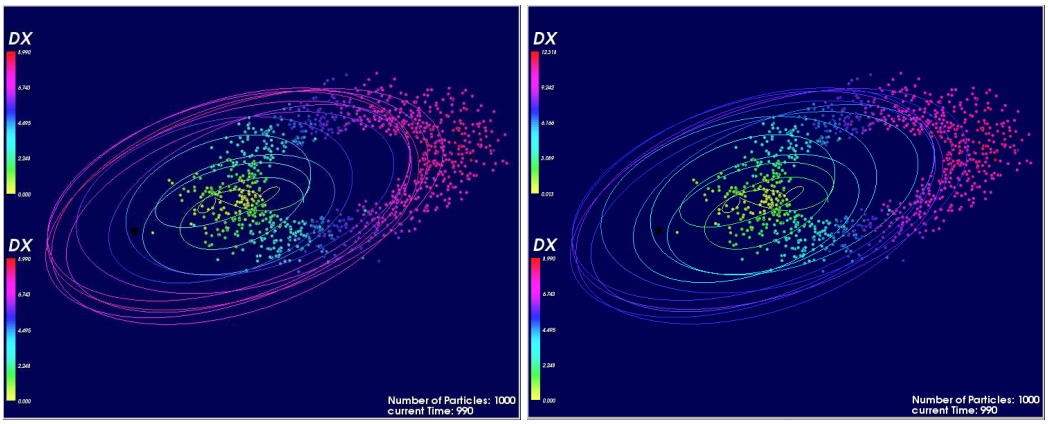


Abbildung 7.5: Visualisierung der Flugbahn eines Partikels: Hier wird die Flugbahn des Partikels mit der Identifikationsnummer 750 angezeigt. In beiden Bildern wurde die Flugbahn anhand der Geschwindigkeit des Partikels eingefärbt. Links wurden als Grundlage der Minimal- und der Maximalwert der Geschwindigkeit aller Partikel zum angezeigten Simulationszeitpunkt ($t = 990$) und rechts zu allen im Zwischenspeicher vorliegenden Datensätzen gewählt.

Kapitel 8

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Entwicklung einer Online-Visualisierung zu dem Simulationsprogramm NBODY6++. Hierzu mussten mehrere Aspekte im Hinblick auf die Kommunikation zwischen Simulation und Visualisierung, die Zwischenspeicherung der Simulationsdaten und die graphische Darstellung der Simulationsdaten berücksichtigt werden.

Zum einen musste die Kommunikation zwischen Simulation und Online-Visualisierung so organisiert werden, dass der dadurch entstehende Overhead bei der Simulation kontrollierbar und beschränkt bleibt. Zum anderen sollten die Kommunikation mit der Simulation und die Benutzerinteraktion simultan stattfinden können.
Zur Lösung des ersten Problems wurde die Kopplungsbibliothek VISIT eingesetzt. Die Online-Visualisierung übernimmt bei der Kommunikation mit der Simulation die Rolle eines Servers und reagiert auf die Anfragen der Simulation. Jegliche Kommunikation wird also von der Simulation initiiert. Zwischenspeicher auf der Simulations- und Visualisierungsseite ermöglichen die Reduktion der Häufigkeit von Übertragungen.
Das zweite Problem wurde mit Hilfe der Event-Queue des GUI-Toolkits Qt gelöst. Kommunikationsereignisse werden zusammen mit anderen Ereignissen (z.B. Benutzerinteraktion) in die Event-Queue eingereiht. Die Ereignisse werden dann nacheinander abgearbeitet.

Eine weitere Aufgabe bei der Entwicklung der Online-Visualisierung war die Implementierung eines Zwischenspeichers für die Simulationsdaten. Hierbei mussten spezielle Eigenschaften der Daten berücksichtigt werden. Aus Gründen der Datenreduktion werden die Partikeldaten nicht zu jedem Zeitpunkt vollständig übertragen. Für die Abfrage von Datensätzen aus dem Zwischenspeicher zu einem Simulationszeitpunkt wurde daher eine Vorausberechnung auf der Basis von früheren Zeitpunkten implementiert.
Bei der Abfrage von Flugbahnen musste die Regularisierung zweier oder auch mehrerer Partikel berücksichtigt werden. Auf diese Besonderheit der Simulationsdaten wurde eingegangen, indem eine Funktion implementiert wurde, welche die zu einer Flugbahn gehörenden Daten rekursiv im Zwischenspeicher sucht.

Die graphische Darstellung der Simulationsdaten stellte einen weiteren Schwerpunkt der Arbeit dar. Hierzu gehörte auch die Integration der Visualisierung in eine graphische Benutzeroberfläche. Die Visualisierung wurde mit Hilfe des Visualisierungstoolkits VTK realisiert. Derzeit können die Positionen und Geschwindigkeiten der Partikel durch Kugeln bzw. Punkte und Pfeile dargestellt werden. Andere Attribute können durch Farbgebung und Skalierung der Objekte dargestellt werden.

Die Online-Visualisierung *xnbody* wurde so entwickelt, dass sie leicht erweiterbar ist. Zum Bei-

spiel können weitere Partikelattribute in die Visualisierung integriert werden; bisher wird nur eine Auswahl von Attributen von der Simulation zur Visualisierung übertragen.
Eine weitere Erweiterungsmöglichkeit von *xnbody* ist die Erzeugung von Filmsequenzen aus der Oberfläche, die zu Präsentationszwecken genutzt werden können.
Auch weitere Interaktionsmöglichkeiten mit der Visualisierung sind denkbar, zum Beispiel die Auswahl einzelner Partikel mit der Maus zur Abfrage von Informationen über das betreffende Partikel oder zur Anzeige von dessen Flugbahn.

Im Rahmen der Zusammenarbeit mit den Anwendern des Simulationsprogramms NBODY6++ werden sich weitere Möglichkeiten der Erweiterung von *xnbody* ergeben.

Anhang A

Nassi-Shneiderman-Diagramme

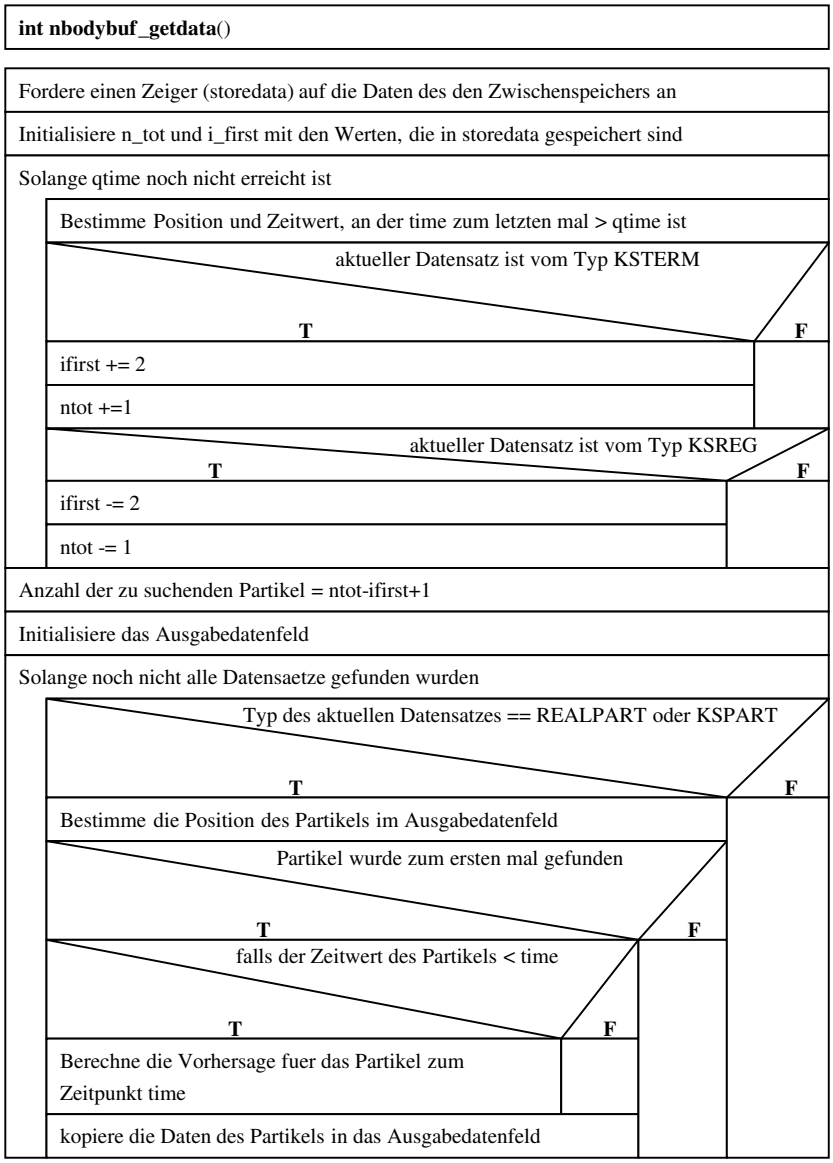


Abbildung A.1: Nassi-Shneiderman-Diagramm der Funktion *nbodybuf_getdata*

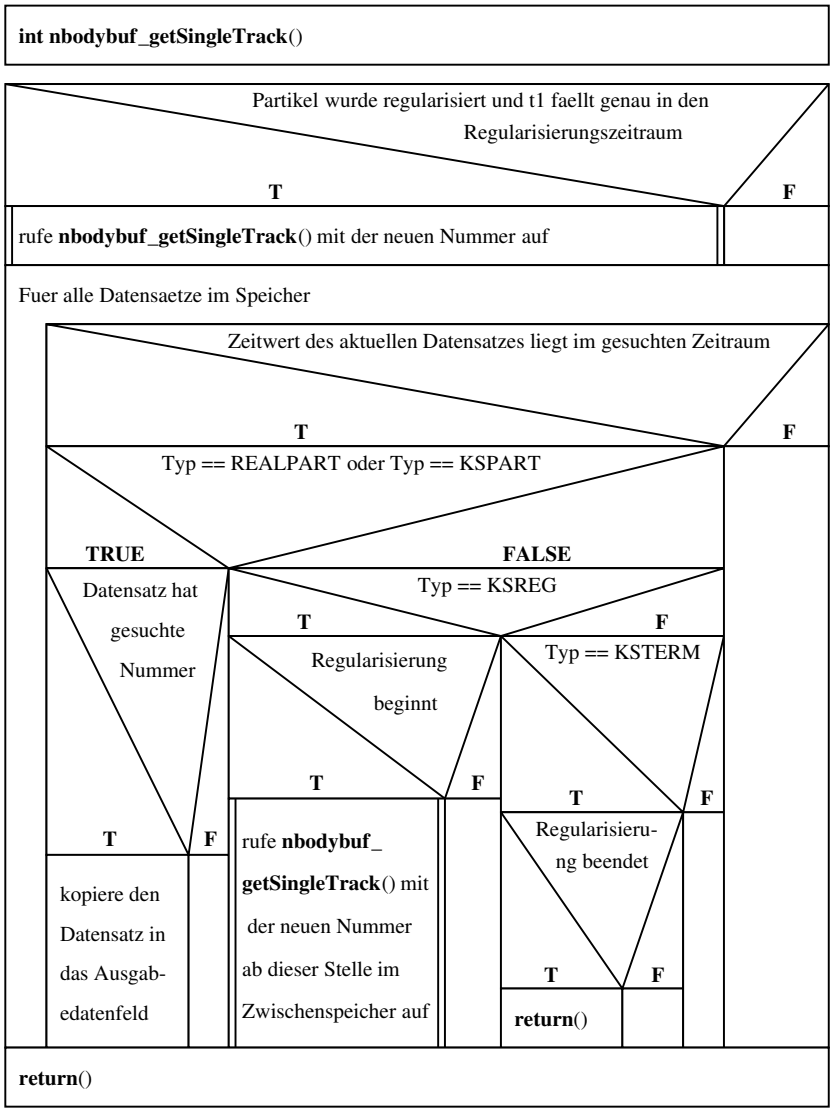


Abbildung A.2: Nassi-Shneiderman-Diagramm der Funktion *nbodybuf_getSingleTrack*

Literaturverzeichnis

[L1] Strategien zur Kopplung und Datenreduktion bei der Online-Visualisierung von parallelen Simulationsrechnungen mit verteilter Datenhaltung – Wolfgang Frings – Jül-4021

[L2] Visit – a Visualization Interface Toolkit, Benutzerhandbuch – Thomas Eickermann, Wolfgang Frings, Anke Häming

[L3] http://www.opengl.org/ – Homepage der Industry's Foundation for High Performance Graphics

[L4] http://public.kitware.com/VTK/what-is-vtk.php – Was ist VTK – Homepage von Kitware, Inc

[L5] The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics and Visualization – William J. Schroeder, Kenneth M. Martin, William E. Lorensen, GE Corporate Research & Development

[L6] The VTK User's Guide – Kitware, Inc.

[L7] http://www.trolltech.com/products/qt/index.html – Qt Produkt Übersicht – Homepage von Trolltech

[L8] Programming with Qt, Matthias Kalle Dalheimer, O'Reilly 2002

[L9] Qt 3.3 Whitepaper, Trolltech (http://www.trolltech.com/products/whitepapers.html)

[L10] NBODY6, Features of the computer-code – Emil Khalisi, Rainer Spurzem (ftp://ftp.ari.uni-heidelberg.de/pub/staff/spurzem/nb6mpi/nbdoc.tar.gz)

[L11] Journal of Computational and Applied Mathematics 109 (1999) 407-432, Direct N-body simulations, Rainer Spurzem

[L12] Gravitational N-Body Simulations, Tools and Algorithms – Sverre J. Aarseth – Cambridge Monographs on Mathematical Physics

[L13] http://wwwipr.ira.uka.de/∼kuebler/vtkqt/ – Homepage von VTK_QT – Carsten Kübler